6.375 Complex Digital Systems Design
Rateless Wireless Networking with Spinal Codes
Final Project Report

Mikhail Volkov, Minjie Chen, Edison Achelengwa

Electrical Engineering and Computer Science
Massachusetts Institute of Technology

May 11, 2011

Contents

1 Introduction

1.1 Project Objective

The aim of this project is to provide an implementation for a rateless wireless networking scheme called Cortex. This scheme was developed quite recently in MIT CSAIL [3]. Rateless networking offers performance benefits over traditional fixed rate coding schemes. The sender encodes the data using a novel rateless *spinal code* which uses a random hash function over the message bits to directly produce a sequence of constellation symbols for transmission. The sender and receiver use a feedback protocol to determine whether a given chunk of data has been correctly decoded.

Practical communication systems rely on data throughput. The very notion of rate only has real meaning within the context of the end product. Testing a communication system in hardware is very important in order to gauge the feasibility of an algorithm in a real system. The goal of this project is to develop a hardware prototype of a Cortex decoder on an FPGA, to investigate the feasibility of the protocol in a practical communication system, and to investigate how much performance improvement can be achieved by optimizing the system for a hardware implementation.

1.2 Overview

A code is a rule for converting a piece of information into another form or representation, not necessarily of the same type. Encoding is the process by which information from a source is converted into symbols to be communicated. Decoding is the reverse process, converting these code symbols back into information understandable by a receiver. A complete communication system works as follows: the encoder receives a message and encodes it into a set of symbols; the symbols are sent across a noisy channel where they may potentially be corrupted; the decoder receives the symbols and decodes them to reconstruct the original message.

Cortex is a rateless communication protocol, that is the sender starts transmitting packet data at a suitably high rate (generally higher than what the channel can currently sustain), and keeps going until the receiver determines that it has correctly decoded all the data and informs the sender to move to the next packet. The Cortex protocol requires that the encoder and decoder agree on a seed s_0, a hash function h and an IQ plane constellation mapping f. The hash function takes two inputs, a real number between $(0, 1)$ and a k-bit message string, and returns another real number between $(0, 1)$. That is

$$h : (0, 1) \times \{0, 1\}^k \rightarrow (0, 1).$$

The constellation mapping f takes as input a stream of $2C$ bits and generates a phasor in the IQ plane.

The system is further parametrized by several values: the message size n, the code step segment size k, and the IQ bit length $2C$ (C for I and C for Q).

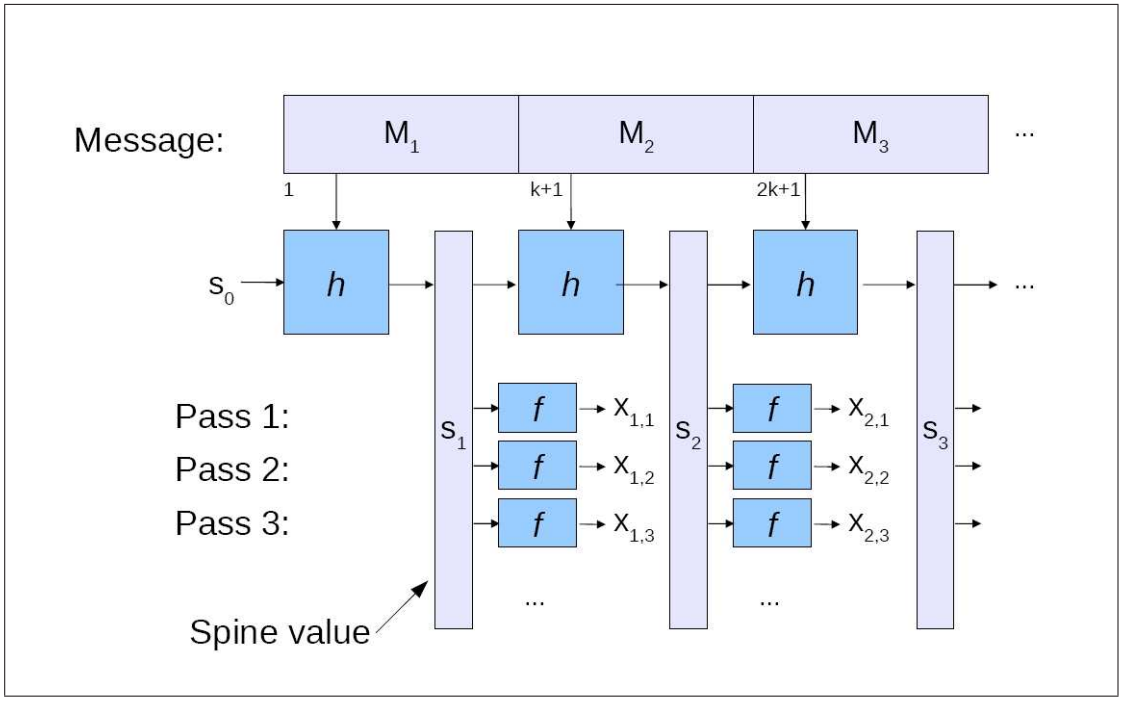

Figure 1: Encoder Overview

1.2.1 Encoder Overview

Figure 1 shows a block diagram for the encoder.

The encoder takes an n-bit message $M = m_1 m_2 \ldots m_n$ and proceeds as follows. First the message is split into n/k k-bit segments M_i. Then it computes n/k spine values

$$s_t = h(s_{t-1}, M_t)$$

using the pre-agreed seed s_0. The encoder then performs passes over the spine as follows. In each pass, n/k new constellation points are generated. In the l-th pass the encoder applies the constellation mapping f to $2C$ bits of the spine starting from position $2C(l-1)$ to generate the t-th constellation point.

The constellation points $x_{t,l}$ are then sent across an AWGN channel.

1.2.2 Decoder Overview

The decoder receives as input a stream of constellation symbols $y_{t,l} = x_{t,l} + w_{t,l}$ where $w_{t,l}$ is the effect of noise and interference.

The decoding process works as follows. The encoder and decoder agreed on h and s_0 *a priori*. The decoder can therefore generate all 2^n possible candidate symbols s_1 using h. In practice, this is not necessary. Each time the decoder receives a symbol

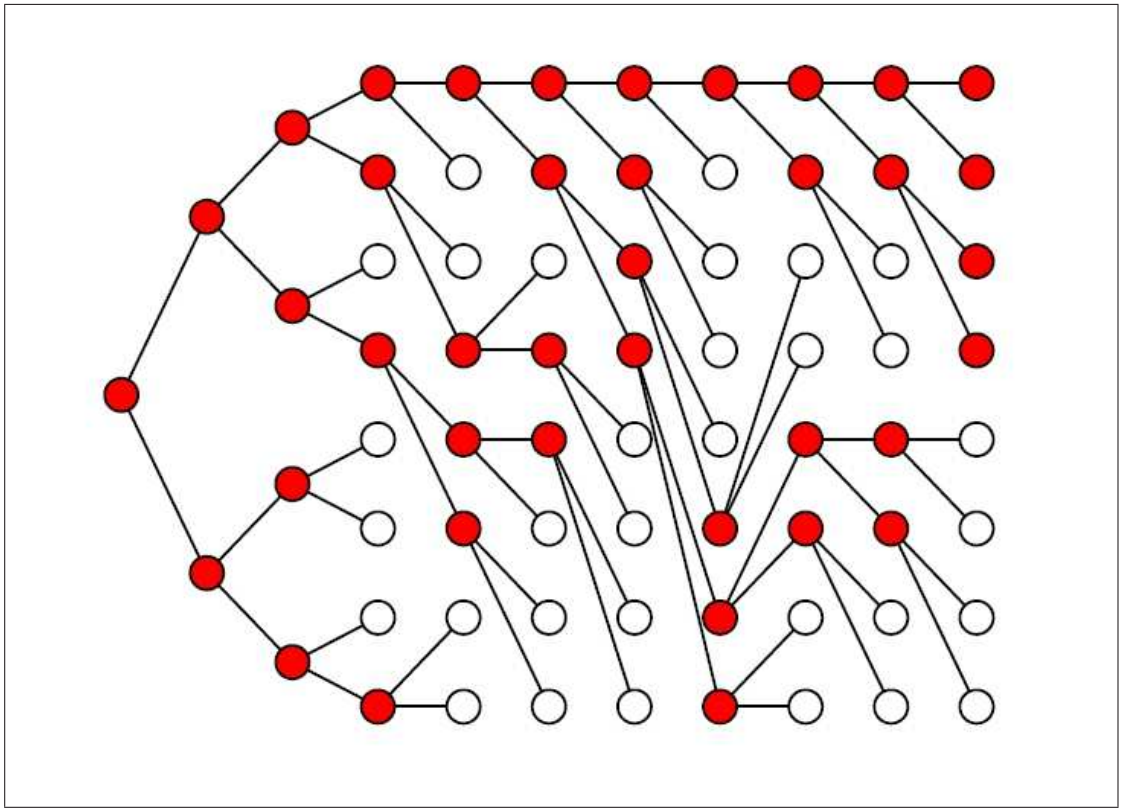

Figure 2: Practical Decoder Tree

$y_{t,l}$ it retains at most B possible nodes from the decoding tree based on the smallest total cost given at each level by $||y_{t,l} - x_{t,l}(s_t)||^2$. Figure 2 shows an example of such a pruned tree. A message is represented by a continuous path from the root node to one of the leaves, and hence there can only be B candidate messages. The decoder estimates the transmitted message as the one that has minimum total cost among the B messages corresponding to the leaf nodes at the level $n = k$ of this pruned tree.

1.2.3 Puncturing Schedule

An important part of the actual implementation of this system is the puncturing schedule. This is agreed-upon by the sender and received *a priori*. In the system, as described above, the receiver obtains symbols corresponding to an integral number of passes. The puncturing schedule allows a much more finer-grained control over the achievable rates. In the first pass, the sender transmits a symbol for every value in the spine sequence. In any subsequent pass, the sender can choose to transmit only every g-th value, where g may change from pass to pass. For example, one option is to use the schedule (8,4,2,1). That is, the first punctured pass sends every 8th value, the second pass sends every 4th value but only if it is not a multiple of 8, the third pass sends every second value but

only if it is not a multiple of 4, the fourth pass sends every value but only if it is not a multiple of 2, repeating this cycle as long as necessary.

Even with a simple schedule, the last symbol of a packet needs to be sent more times than the previous ones. This is due to the nature of the algorithm being such that the errors are most concentrated at the leaves of the pruned decoder tree. The project implements an infrastructure for specifying an arbitrary puncturing schedule

The decoder must know the puncturing schedule in advance, as it must know which symbol in the decoding tree a given received symbol is supposed to be compared against.

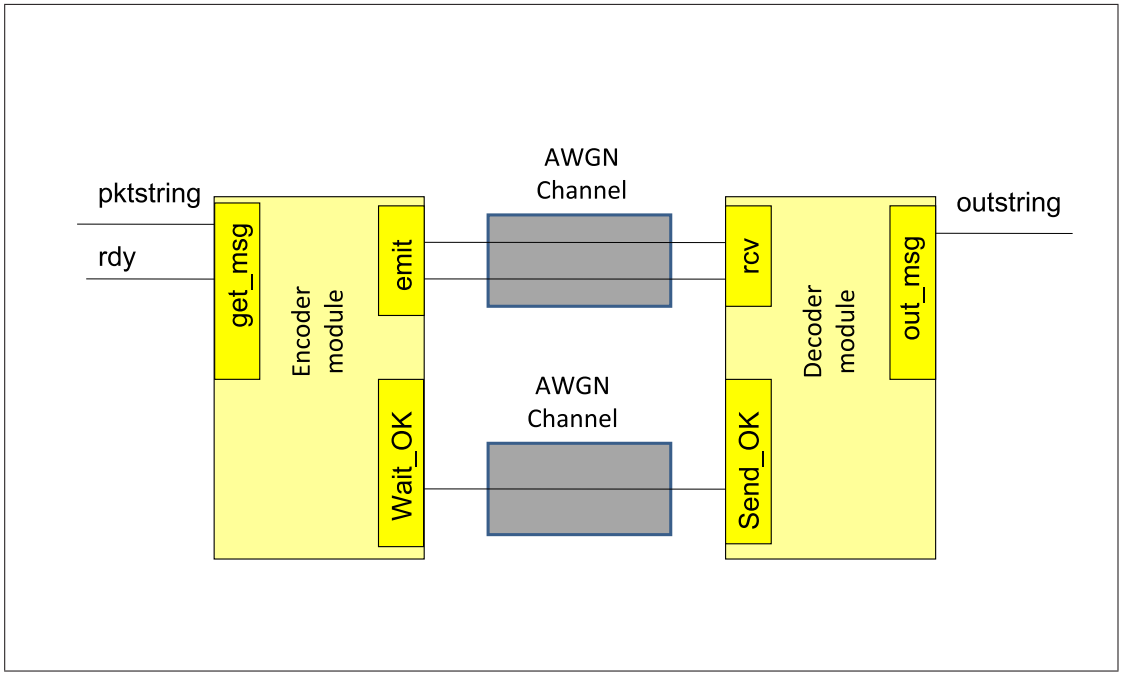

Figure 3: High-level system architecture

2 High-Level Design and Test Plan

The high-level system architecture is shown in Figure 3. The decoder stages and rules are shown in Figure 4. The diagram shows the finalized module interaction.

2.1 Interface

The Decoder is a server interface which takes as input a symbol (of type `Symbol`) and outputs a message (of type `Msg`) which is a 192-bit number.

2.2 Sub-modules

The first sub-module is the Puncturing Scheduler. It is initialized with the same parameters as those in the Encoder. There is potential for polymorphism in the sense that different types of scheduling can be achieved through the Puncturing Scheduler.

The second sub-module is the Spine Evaluator. The Spine Evaluator contains both implementations of the hash function, $h(*)$, (i.e. Salsa) and the Symbol Mapper function, $f(*)$.

The final sub-module is the Sorter. This module will be used when the suggestions of the potential correct code for a code step are sorted according to costs and the best B ones (where B = beam width) are chosen and stored in `BackTrackMem` which represents the tree.

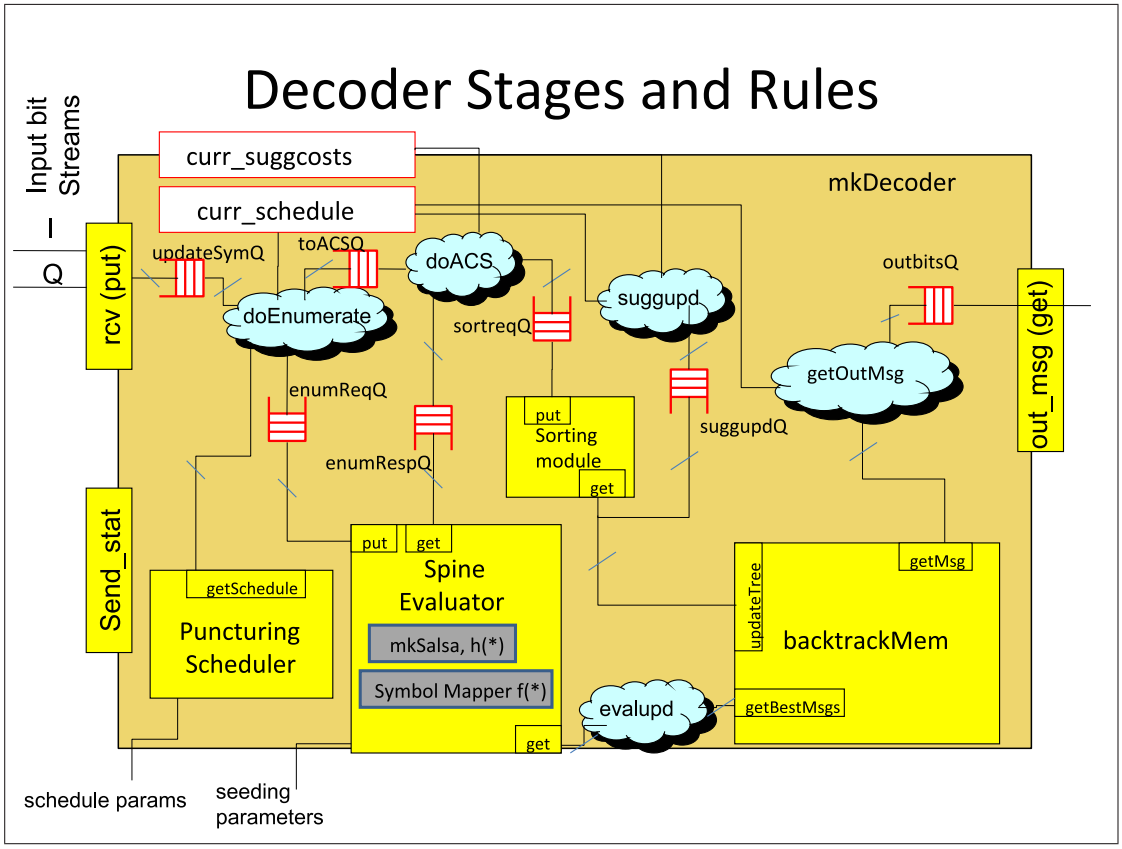

Figure 4: Decoder stages and rules

2.3 States

The two states are `curr_suggestions` and `curr_schedule`. The `curr_schedule` state keeps track of what code step and corresponding symbol index the encoder is taking care of. The symbol index state is kept in `curr_schedule` because of the last code step for which more than one symbol is sent. `curr_suggcosts` keeps track of the maximum likelihood costs of the leaves of the tree structure of the decoder algorithm. The suggestions are always of size B or less and they are the best costs from the previous code step. These states are used by the `doACS` stage and updated by the `suggupd` stage which is also when the `BackTrackMem` module is getting its updates from the Sorter.

The last code step is treated differently from the other code steps. Since error is accumulated as we approach the last code step, more symbols are sent for the last code step. There is another state which the decoder uses called `laststepcosts` which is a vector $B2^k$ costs. Since multiple symbols are sent for the last code step, instead of the algorithm sorting after the costs of the first symbol index is processed it accumulates costs for all the update symbols of the last code step before sorting them.

2.4 Stages and Rules

The stages of this algorithm were modeled from the Python decoder algorithm. When a symbol is received it is placed in the `updateSymQ` FIFO, here are the different things which happen in different stages (rules).

As far as the concurrency is concerned, the stages were guarded such that only one fires at a time. This was supposed to be the first round of the implementation for correctness of the algorithm. Generally the stages occur in this order: `doEnumerate`, `doACS`, `suggupd` and `evalUpd`. Then at the last schedule (last symbol index of the last code step) only the `getOutMsg` stage occurs. The order is maintained except for intermediate symbol indices of the final code step. Here only `doEnumerate` and `do ACS` happen until the algorithm processes the last symbol index of the last code step in which case it sends the costs to the Sorter and goes to `suggupd` stage and `evalUpd`.

2.4.1 Code Enumeration

The code enumeration stage (`doEnumerate`) gets a schedule from the Puncturing Scheduler and the schedule gives information about what symbol index is being treated and what code step is being treated. Using these, it sends a request to the Spine Evaluator, through the `enumReqQ` FIFO, to compute the all possible next spine values and all possible `Symbol` which could correspond to the new symbol. The request is defined by the code step and the symbol index. It should be noted that we always keep B best suggestions from the previous code step. This means somehow the Spine Evaluator should keep B best spine values and compute $B \times 2^k$ possible new Spine values and consequently $B \times 2^k$ possible new symbols based on the Symbol Index and Code Step of the new schedule. Unfortunately the Spine Evaluator cannot know which are the best B spines until the `doACS` stage computes the likelihood costs and the Sorter sorts the costs and gets the best B ones. Consequently at this stage the Spine Evaluator does not store any of the spines but directly calculates the possible symbols. It will get its updates at a later stage called `EvalUpd`. Also, during the `doEnumerate` stage, the new update symbol from the `updateSymbQ` FIFO is queued into another FIFO called `toACS` which will be used by the `doACS` stage for ML costs.

2.4.2 Add-Compare-Select

The ACS stage (`doACS`) happens after `doEnumerate`. `doACS` takes the results from `enumRespQ` which is a vector of $B \times 2^k$ symbols and the new update symbol from the `toACSQ` FIFO and uses these to computes the costs of each of the $B \times 2^k$ suggested symbols as follows: $cost = parent\_cost + (update\_symbol - possible\_symbol)^2$. The parent cost is obtained from a state kept in the decoder called `curr_suggcosts` which is a vector of B best costs. The Spine Evaluator sends its symbols in a particular order in the `enumRespQ` FIFO: the first 2^k symbols are computed from the first parent then the next from the next parent and so on till the B-th parent. For each parent, its 2^k symbols correspond to codes which vary in increasing order from 0 to $(2^k) - 1$. Hence the likelihood costs before being sent to the Sorter are marked in order from 0 to $(B \times 2^k) - 1$

as the Spine Evaluator sent the symbols. Then these vector of $B \times 2^k$ marked costs are sent into the Sorter to be get the best B ones.

2.4.3 Suggestion Update

The suggestion update stage (`suggupd`) happens after the `doACS` stage. During this stage the new best B marked costs are obtained from the Sorter and they are sent to `BackTrackMem` to update its best B message prefixes accordingly. Also, during this stage the `curr_suggcosts` state is updated to be used by `doACS` for the next state.

2.4.4 Spine Evaluator Update

The spine evaluator update stage (`evalUpd`) only happens after `BackTrackMem` has updated itself. Then it sends the best marks from the Sorter to the Spine Evaluator. It is here that the Spine Evaluator now computes the best B spines and updates itself for the next `doEnumerate` stage.

2.4.5 Get Output Message

Finally, the stage that retrieves the output message (`getOutMsg`) can only happen when the decoder is ready. The decoder is ready when the final symbol of the final code step has been processed and `BackTrackMem` has updated its `bestmsgs` accordingly. This stage makes the output decoded message available to the outside world.

2.5 Test Plan

The test plan was to test the decoder against the decoder algorithm implemented in Python. A message string is to be encoded to a set of symbols based on a puncturing schedule. Then the Python decoder takes those symbols from the encoder and using the same puncturing schedule is expected to output the right message.

Hence it was important to get test case symbols. These test case symbols was obtained from taking the string which when encoded into hex gives a 192-bit number. The 24-character string "this string has 24 chars" was used. Then the Python encoder encodes the message and creates a stream of symbols. A test-bench was written which instantiates a decoder, creates one input vector which will be the vector of the symbols and continuously feeding those symbols into the decoder and then when the last symbol has been fed, the decoder is queried for a decoded message.

Comparisons were made between the behavior of our decoder and the Python decoder. The debugging process was guided by this algorithm. Also the test bench implemented certain helper functions for transforming the 192 bit received message from the decoder to the original string.

The implementation and testing plan was broken down into 5 stages:

1. First, a decoder backbone was created with dummy modules. This provided us with high-level Bluespec scaffolding and allowed us to concentrate on independent

details of the decoder in isolation. The Puncturing Scheduler and Backtrack Memory were developed in situ as they were relatively primitive in comparison with the other modules. The Hash module, the Spine Evaluator and the Sorter were developed separately and tested independently.

2. The Sorter was developed and tested independently. The dummy sorter used in the development of the decoder backbone simply was simply designed to output the first B symbols of the $B \times 2^k$ that it received.

3. The Salsa module was developed and tested independently. Two different initial states were chosen as reference states: `0` and `0x0a0b0c0d01020304`. Correctness was the immediate priority during the first implementation of Salsa, and the algorithm was replicated directly according to the Salsa specification in [1]. The Symbol Mapper was then implemented. The rest of the system is not aware of the existence of the Hash module or of the Symbol Mapper, as it interfaces only to the Spine Evaluator, thus these two components were developed and tested independently without the risk of jeopardizing the correctness of the rest of the decoder. Having implemented and tested the Salsa module and the Symbol Mapper, the Spine Evaluator was developed. This was tested extensively through comparison with existing Python code, using a separate testbench. In the meantime, the dummy spine evaluator would simply output $B \times 2^k$ zeros in accordance with its interface specification.

4. With the Sorter, Spine Evaluator and Decoder Backbone developed and tested independently, the crucial moment came when these modules were put together into the first version of a complete decoder. The goal here was simply to ensure that bits were flowing end-to-end and that the micro-architectural correctness was maintained. This is where using a language such as Bluespec helped immensely. With its strong types and strict interface semantics, once the modules were developed and tested separately, plugging them in was a formality. Everything worked on the first try, although (as expected) the output message was not being decoded correctly.

5. Finally, the most important stage of testing was to ensure that the decoder actually produced correct results. With all modules functioning correctly at an architectural level, correctness at a semantic level was achieved by meticulous bit-by-bit debugging. Bluespec's simulation facilities were employed to their full potential, and through a long and arduous process of displaying values, comparing with reference code, locating and fixing problem, rinsing and repeating all bugs were eliminated.

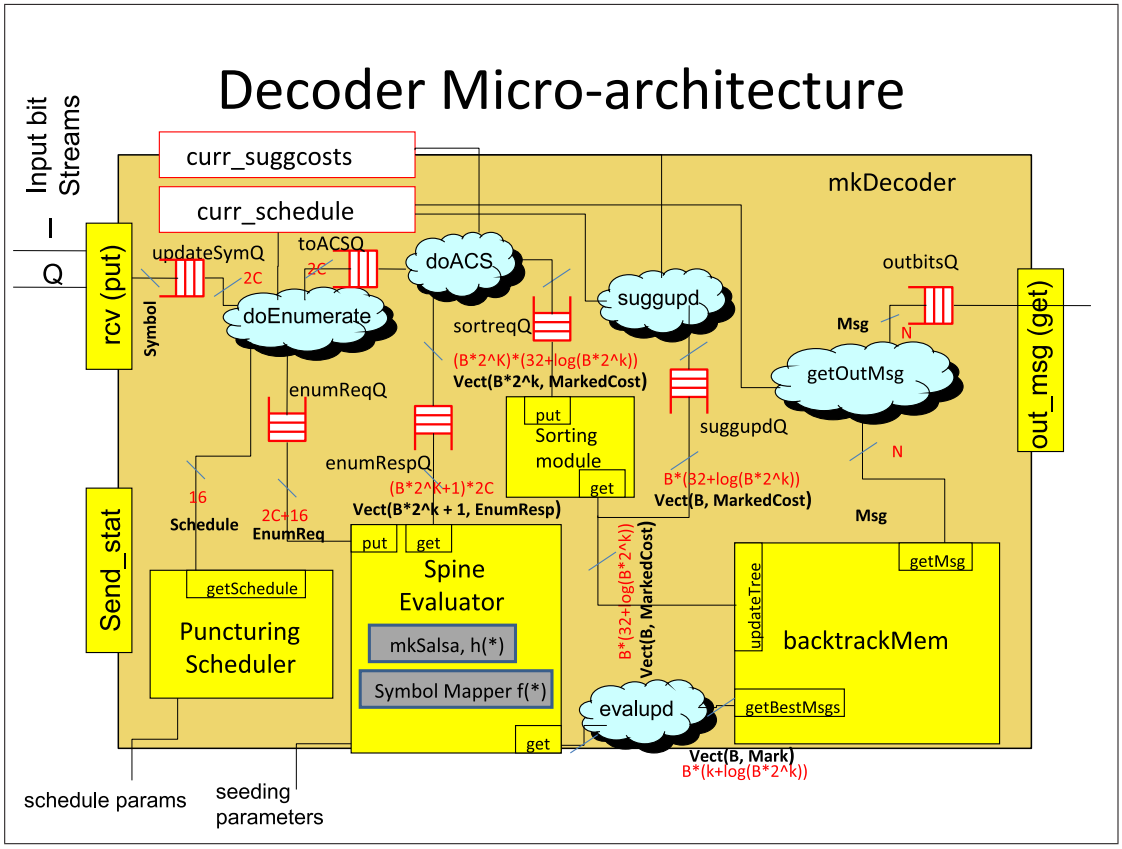

Figure 5: Decoder micro-architecture

3 Micro-Architectural Design

The decoder micro-architecture is shown in Figure 5. The diagram shows the finalized implementation interfaces and module interaction.

3.1 Puncturing Scheduler

In the original algorithm, the encoder and the decoder agree on a schedule of how to send the symbols. The schedule dictates how the next symbol is obtained. A schedule is defined by two numbers, a code step and a symbol index. The code step varies from 1 to n/k and the symbol index is the index of the section of the spine from which the symbol is obtained. This schedule is necessary for the decoder to know so that the decoder can do the ML comparison with the right set of symbols.

In the decoder the puncturing schedule is determined by a sub-module called the Puncturing Scheduler. This scheduler has only one main method `getSchedule` that returns a type called `Schedule` which is a struct with `Codestep` and `SymbolIndex` fields. Calling this method will return the next schedule according to the scheduler module.

The module is initialized by 1 parameter called `NumSymbForLastCodeStep` which is the number of symbols the decoder should be expecting for the last code step. The reason why more symbols are sent for the last code step is that if symbols are sent through a noisy channel then the error is accumulated and the last code step is the most error-prone. Hence more symbols are sent for the last code step.

The Puncturing Scheduler is a relatively simple module with two states called `curr_codestep` and `curr_symbInd` which are the current code step and the current symbol index and both are initialized to 0 at the creation of the scheduler. Every time the `getSchedule` method is called `curr_codestep` and `curr_symbInd` are used to instantiate a `Schedule` object that is returned. If `curr_codestep` is not the last code step, `curr_codestep` is incremented, else if `curr_codestep` is the last code step and the symbol index is less than the last symbol index of the last code step, the symbol index is incremented. If none of the above conditions are fulfilled then both the code step and symbol index are reinitialized to 0.

The above Puncturing Scheduler works for one pass of decoding.

3.2 Salsa Hash Module

We first introduce the Salsa20 standard and then explain the hardware implementation.

3.2.1 Salsa20 Standard

The Salsa20 standard is presented in [1]. Essentially, Salsa20 can be viewed as

$$Salsa20 : 512\text{bits} \longrightarrow 512\text{bits} \tag{1}$$

Salsa20 uses three primitive operations that make it particularly well-suited to hardware implementations: addition, exclusive-or and rotation. Salsa20 defines several basic functions that use these primitive operations.

If b_i is a byte and x_i, y_i are 32-bit words, then:

$quarterround : (x_0, x_1, x_2, x_3) \longrightarrow (y_0, y_1, y_2, y_3)$ where

$$\begin{aligned} y_1 &= x_1 \oplus ((x_0 + x_3) \lll 7), \\ y_2 &= x_2 \oplus ((y_1 + x_0) \lll 9), \\ y_3 &= x_3 \oplus ((y_2 + y_1) \lll 13), \\ y_0 &= x_0 \oplus ((y_3 + y_2) \lll 18). \end{aligned} \tag{2}$$

$rowround : (x_0, x_1, \ldots, x_{15}) \longrightarrow (y_0, y_1, \ldots, y_{15})$ where

$$\begin{aligned} (y_0, y_1, y_2, y_3) &= quarterround(x_0, x_1, x_2, x_3), \\ (y_5, y_6, y_7, y_4) &= quarterround(x_5, x_6, x_7, x_4), \\ (y_{10}, y_{11}, y_8, y_9) &= quarterround(x_{10}, x_{11}, x_8, x_9), \\ (y_{15}, y_{12}, y_{13}, y_{14}) &= quarterround(x_{15}, x_{12}, x_{13}, x_{14}). \end{aligned} \tag{3}$$

$columnround : (x_0, x_1, \ldots, x_{15}) \longrightarrow (y_0, y_1, \ldots, y_{15})$ where

$$\begin{aligned}(y_0, y_4, y_8, y_{12}) &= quarterround(x_0, x_4, x_8, x_{12}),\\ (y_5, y_9, y_{13}, y_1) &= quarterround(x_5, x_9, x_{13}, x_1),\\ (y_{10}, y_{14}, y_{12}, y_6) &= quarterround(x_{10}, x_{14}, x_2, x_6),\\ (y_{15}, y_3, y_7, y_{11}) &= quarterround(x_{15}, x_3, x_7, x_{11}).\end{aligned} \tag{4}$$

$doubleround : (x_0, x_1, \ldots, x_{15}) \longrightarrow (y_0, y_1, \ldots, y_{15})$ where

$$doubleround(x) = rowround(columnround(x)). \tag{5}$$

$Salsa20 : (b_0, b_1, \ldots, b_{63}) \longrightarrow (b'_0, b'_1, \ldots, b'_{63})$ where

$$Salsa20(b) = b + doubleround^{10}(b) \tag{6}$$

where each 4-byte sequence is viewed as a word in little-endian form.

3.2.2 Expansion Functions

The expansion functions allow us to define Salsa20 as

$$\begin{aligned}Salsa20 : 256\text{bits} &\longrightarrow 512\text{bits}\\ \text{and}&\\ Salsa20 : 384\text{bits} &\longrightarrow 512\text{bits}.\end{aligned} \tag{7}$$

The cortex algorithm uses the first form. To this end we define a constant $\tau =$ "expand 16-byte k" in ASCII. Then define

$$Salsa20_k(n) = Salsa20(\tau_0, k, \tau_1, n, \tau_2, k, \tau_3). \tag{8}$$

3.2.3 Salsa Implementation

In principle, we wish all our spine values to be infinite-precision floating point numbers as shown in Figure 1. In practice the spine values are represented as 64-bit long integers, each mapped to a possibly infinite number of symbols. At each step, we take the previous spine s_{t-1} and a block M_t of k bits, giving $64 + k$ bits total and inflate them to produce a 256- or 384- bit input to Salsa20. We then take the output bits to create the next spine and a few symbols. More bits are then produced as necessary by re-applying Salsa20 in a certain way.

Figure 6 shows the micro-architecture for the `mkSalsa` module. Each Salsa20 block performs the Salsa20 expansion function as presented above. The difference between the architecture as shown in Figure 1 is that the output is both the next spine value and an update buffer, which is a multiple of 512 bits. The next spine value is used to

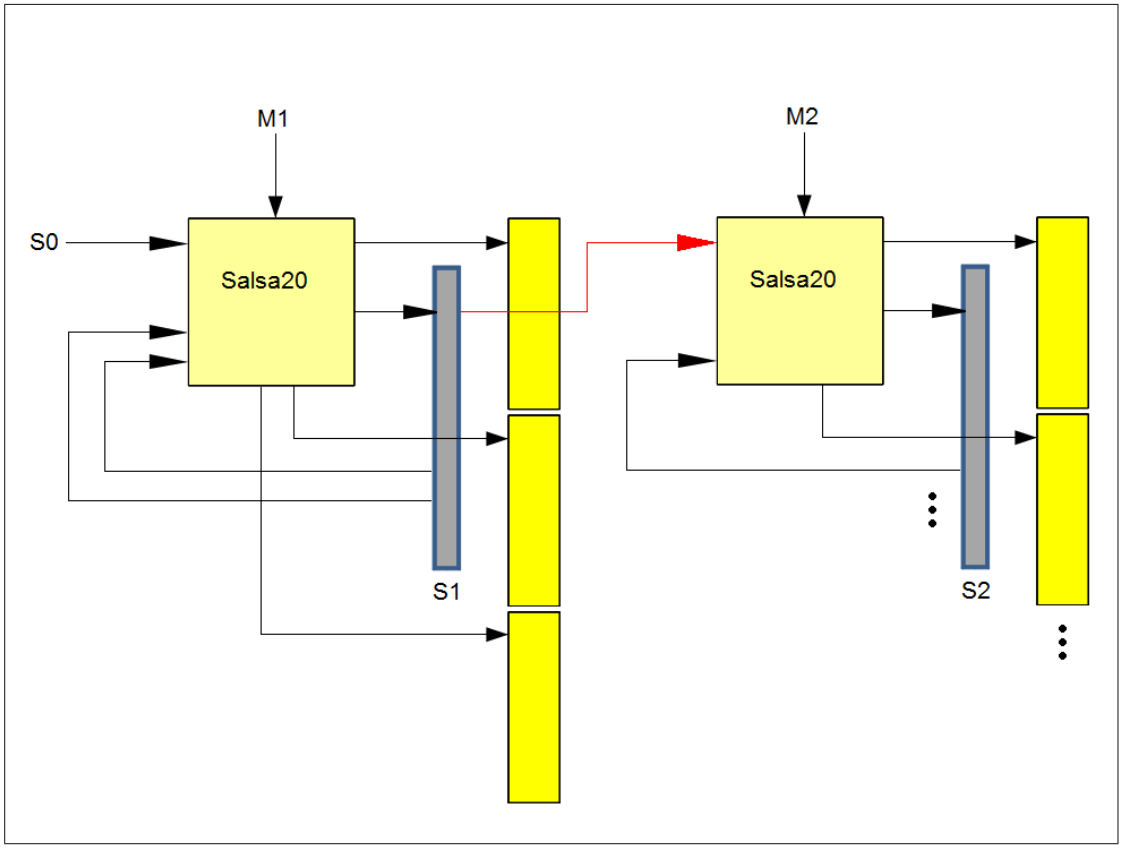

Figure 6: Salsa Hash Micro-architecture

perform the next round, while the update buffer is used to generate the symbols. For this project an assumption was made that SNR is sufficiently high such that no more than 512 output bits would be required to generate all necessary symbols. In theory, with $2C = 10$ this translates into $512/10 = 51$ symbols. However, due to the way Salsa handles its output bits (using 8 64-bit numbers rather than a single 512-bit number), in practice this means 48 symbols. This is more than sufficient for a single code step, and translates into a reasonable SNR assumption.

Applying the hash function in the spinal code setting does not require cryptographic guarantees, so Cortex uses fewer rounds (12) for better performance. Our implementation similarly uses 12 rounds. This reduces (6) to

$$Salsa20(b) \quad = \quad b + doubleround^6(b). \tag{9}$$

3.2.4 Salsa Module Interface

The `mkSalsa` module operates as a black box within the Spine Evaluator. `mkSalsa` will be a server where the requests are a code step and the symbols index, and the response is $2C$ bits from the update buffer corresponding the given request. The spine evaluator then applies the symbol mapper to the `mkSalsa` response and produces the desired symbol. It is likely that some or all of the storage required for the `mkSalsa` state be kept in memory.

3.3 Symbol Mapper

The Symbol Mapper component translates Salsa output bits and a specific symbol index to an actual output symbol. The Symbol Mapper is in-fact a simple combinational function that is unrolled at compile time. One interesting issue with the symbol mapper is that is requires division and modulo by a number that is not a power of 2. This posed problems during synthesis. A simple solution to this problem made use of our assumption that no more than 48 output symbols would be required for a given code step. Rather than using division and modulo, a giant case statement was used to unroll a clause of combinational logic that determined the resulting circuitry at compile time. No generality is lost with regards to polymorphism since we use the power of Bluespec's preprocessor to calculate the necessary division and modulo. Sometimes the simplest solution is truly the most elegant.

3.4 Spine Evaluator

The Spine Evaluator performs two functions. First, the Spine Evaluator must take a request (an `EnumReq` object consisting of a code step and a symbol index) and provide a response (an `EnumResp` object consisting of a vector of $B \times 2^k$ corresponding symbols). Second, the Spine Evaluator must update its spine values so that subsequent requests are evaluated correctly (since Salsa requires the previous hash state as input in order to produce the next hash state).

The Salsa module generates output symbols and a new hash state for each enumerated code that is fed into it. After generating $B \times 2^k$ symbols, only the B best hash states are kept as the updated spine values. The Spine Evaluator instantiates a single Salsa module with which it communicates via its Server interface as described in Section 3.2.4. In order to achieve a feasible Spine Evaluator implementation, the two functions outlined above are performed separately.

First, the Spine Evaluator enumerates all 2^k codes B times and generates the required $B \times 2^k$ symbols by querying Salsa (and then Symbol Mapper) $B \times 2^k$ times. However, at this stage the new spine values are discarded and only the output symbols are used to populate the vector for the `EnumResp` response. The response is then sent out, the decoder performs the necessary ACS and sorting, and the Backtrack Memory has now recorded the B best current messages.

Second, the Spine Evaluator receives an update from the Backtrack Memory in the form of the B best marks from the $B \times 2^k$ enumerated symbols. The symbols are

enumerated by code and then by spine index, thus performing division and modulo by 2^k retrieves both the code and spine which was used to generate these B best symbols. Using these B codes and corresponding spine indices, the Spine Evaluator queries Salsa once again, but this time discarding the symbols and storing the resulting spine values. Finally, these spine values are stored as the current spine values and the Spine Evaluator advances to the next code step.

If the code step in the current request is the same as the current code step in the Spine Evaluator, then no Salsa query is necessary and the symbols from the previous round are returned. In this case no spine update occurs either.

3.5 Sorter

The Sorter module has be built based on a core type called `MarkedCost`, which can be implemented as a struct with two components: `mark` and `cost`. `mark` is used to identify its parent symbols, and `cost` is the vector distance of the predicted symbols compared to the received symbols. As described in the higher-level architecture, the Sorter module receives a vector of data in the format of `MarkedType`, find B codes with the smallest `cost`, and then output it in the format of another vector of `MarkedType` with size B.

When implementing it with Bluespec, we use `Server#` as the external interface. Firstly, the incoming vector of `MarkedCost` is put into a FIFO called `inputFIFO`, and a FIFO called `outputFIFO` is created. The incoming vector is then put into a register called `inputReg`. We designed a function called `findtheBestandUpdate`, which takes in a new struct called `TaggedMarkedCost` with two components, `costVector` and `mincost`: `costVector` is a vector of Maybe#(MarkedCost). The reason why we use Maybe# is that in the function `findthebestandUpdate` we are going to mark the code which has already been selected with `Invalid`; `mincost` is a data in the type of `MarkedCost`, which stores the selected best `MarkedCost`: it has the smallest `cost` and its corresponding `mark`. Then an iterative process begins. We placed the value stored in the `inputReg` into the function of `findthebestandUpdate` and get the selected `MarketCost`, and the corresponding position in the `costVector` is tagged as `Invalid`. The selected code with its cost is then stored in another register called `outputReg`. After this, we do `fintheBestandUpdate` on the `inputReg` again, get the second `best` codes, and then store it in the `outputReg`. Doing this iteration B times, we get the best B of the `MarkedCost` from a vector of `MarkedCost` with the size $B \times 2^k$.

In order to test this module, we just set up the test-bench, serve this module with a vector of `cost` and a parameter called `BeamSize`, and then check whether it gives us the best B of `MarkedCost`. We have built and tested the MATLAB implementation of this algorithm, and built the test bench for this module. The Sorter module is tested and fully functional.

3.6 Backtrack Memory

The name `BackTrackMem` comes from the fact that in the algorithm, the possibilities of the messages are represented as a tree and the most likely message is obtained by backtracking through the tree from the final code that has the minimum likelihood cost. However due to the simplified version of the algorithm which assumes only one pass of symbols are passed and are enough, the messages are not represented as trees but the full message prefixes are accumulated in `BackTrackMem` until the last symbol of the last code step is processed, at which point the best B messages are present in the module.

`BackTrackMem` receives updates from the Sorter for every code step processing. The updates are vectors of B `MarkedCost` objects. A `MarkedCost` is a helper type which has two fields which are a likelihood cost and a mark. A mark, based on this algorithm will vary from 0 to $(B \times 2^k) - 1$. It was decided during the design space exploration that when `BackTrackMem` is updating its message prefixes, it needs to know two things: what the new code (child) is and what message prefix (parent) it needs to attach it to. The new code is obtained from the mark by taking remainder of division by 2^k and the message (prefix) position is obtained from the mark by taking the quotient of the division by 2^k. `BackTrackMem` uses the updates from the Sorter to update its two states called `bestmsgs` and `bestmarks`.

`BackTrackMem` has 2 states. The first state, `bestmsgs` is a vector of B messages which by the end of the decoding process will become the best B messages. The second state is called `bestmarks` which is vector of best B `Mark` objects. A `Mark` is a counter which is used to mark symbols as they are being tested for likelihood with the input symbol and consequently it is used to mark their likelihood costs before they are sorted. The state, `bestmarks` is necessary because when the Spine Evaluator computes new symbols it does not store the spine values which are to be used for the next schedule. `BackTrackMem` has a method `bestmarks` which returns the best marks and these are sent to the evaluator which in turn updates itself to its new spine values.

Finally, `BackTrackMem` has a method which returns the best message, which is the first message of `bestmsgs` state.

4 Implementation Evaluation

4.1 Debugging

The following section provides a brief outline of the bugs that we encountered during the bit-by-bit debugging process. These may provide an interesting insight into common mistakes that are easy to make during coding and difficult to debug. The bugs are listed in the order in which they were encountered and fixed.

1. Spine Evaluator (major bug): the two rules that are responsible for the spine update worked in parallel. What happened was the update overwrote current spine values while they were possibly still being fed into the hash module to obtain the new spine values. Thus, multiple updates could occur and caused incorrect spine values to be calculated.

2. Decoder, current suggestion update (minor bug): in current suggestion update rule we iterated from $0 - (B - 1)$ instead of $0 - B$.

3. Spine Evaluator (major bug): when determining how many valid output symbols should be sent we must compare $(2^k)^c$ with $B \times 2^k$ where c is the current code step. The smaller of these quantities is the number of valid symbols that are to be sent. For example if $B = 4, k = 3$ and $c = 1$ then only 8 valid symbols are sent, and the other 24 are invalid. When checking this inequality a register was used, but the quantity $(2^k)^c$ quickly grows larger than the register which held it. At $c = 11$ an overflow occurred and since we were using shifts to calculate powers of 2, the number simply shifted off the register and became equal to 0. Thus all symbols were incorrectly invalidated since $(2^k)^c = 0$ and a major error occurred.

4. Spine Evaluator (minor bug): skipping the second stage of the spine evaluator when the incoming request code step is the same as the current code step caused an incorrect symbol index to be invoked, since the symbol index register was not being updated.

5. Spine evaluator (major bug): spine evaluator needs to be updated after sorting happens with new hashes to be used for the next code step, but spine updates actually did occur during the last code step. Due to this when the first symbol for the last code step was processed, the update was sent and a set of new hashes now exists in the evaluator. But to process the next symbols of the last code step, we are not supposed to use the newly updated symbols, we are supposed to use the symbols for the penultimate code step. To fix this problem, we set the spine evaluator to only accept updates for code steps other than the last one.

6. Decoder (major bug): updates kept being enqueued to the one of the input queues of the spine evaluator. Since the spine evaluator stopped accepting updates for the last code step, at some point the simulation got stuck. This was fixed by also not sending updates for the last code step.

4.2 FPGA Synthesis

After synthesizing the entire decoder module on the FPGA, we are able to get the results of each sub-module. Based on this, we can get an idea of the future optimization of our algorithm and hardware implementation. We implemented two versions of Salsa module which shows us the differences between combinational Salsa and multi-stage Salsa (hereafter referred to as Salsa1 and Salsa2, respectively). We also compared performance against a theoretical hash module which takes 1 cycle (referred to as Salsa0). The entire synthesized results are shown below:

Number of:	Salsa0		Salsa1		Salsa2	
Slice Registers:	18,094/69120	26%	19,150/69,120	27%	20379/69120	29%
Slice LUTs:	20498/69120	29%	23,604/69,120	34%	24200/69120	35%
Used as logic:	19762/69120	28%	22,767/69,120	32%	23389/69120	33%
Used as Memory:	719/17920	4%	807/17,920	4%	811/17920	4%
Bonded IOBs:	11/640	1%	11/640	1%	16/640	2%
BlockRAM/FIFO:	12/148	8%	12/148	8%	11/148	7%
BUFG/BUFGCTRLs:	8/32	25%	8/32	25%	16/32	50%
DSP48Es:	32/64	50%	32/64	50%	32/64	50%
PLLADVs:	2/6	33%	2/6	33%	2/6	33%

Table 1: Device Utilization Summary of Decoder

According to the synthesizing results as shown in Table 1, we can see that Sorter module takes the most area compared to other sub-modules. Within the Sorter module, the Searcher submodule takes the main part since actually, the Sorter module is a repeating case of Searcher module. Currently our Sorter module is combinational, and in future implementations we may be able to pipeline it to lower the area being used.

Salsa1 and Salsa2 do not differ significantly in terms of area. This is because the additional combinational circuitry that was required to unroll the *rowround* and *columnround* operations was not significant. This indicates that further combinational improvements can be made to Salsa in future implementations.

	Salsa0		
Name of Module	Slices	Slice Reg	LUTs
mkBridge (Decoder)	1727/8881	1504/18094	3125/20498
Backtrackmem	192/217	760/806	749/785
SpineEvaluator	368/720	414/1676	571/1496
Salsa	1/148	0/526	1/537
Puncscheduler	8/8	13/13	19/19
Sorter	354/2603	1375/8820	1387/7070
Searcher	638/1584	1271/5017	1912/4413

Table 2: Utilization by Hierarchy of Decoder with Salsa0

	Salsa1		
Name of Module	Slices	Slice Reg	LUTs
mkBridge (Decoder)	2098/9896	1504/19150	3460/23604
Backtrackmem	192/229	760/846	749/809
SpineEvaluator	464/1526	670/2692	776/4223
Salsa	653/849	589/1246	2362/3035
Puncscheduler	7/7	13/13	19/19
Sorter	355/2553	1375/8820	1387/7070
Searcher	622/1566	1271/5017	1912/4413

Table 3: Utilization by Hierarchy of Decoder with Salsa1

	Salsa2		
Name of Module	Slices	Slice Reg	LUTs
mkBridge (Decoder)	2101/10092	1504/19148	3460/23911
Backtrackmem	192/232	760/846	749/809
SpineEvaluator	498/1682	670/2690	777/4530
Salsa	771/968	587/1244	2668/3341
Puncscheduler	8/8	13/13	19/19
Sorter	353/2575	1375/8820	1387/7070
Searcher	641/1589	1271/5017	1912/4413

Table 4: Utilization by Hierarchy of Decoder with Salsa2

4.3 Code Analysis

Module	Lines of source code
DecoderTypes	136
Misc	10
SymbolMapper	267
Decoder	345
DecoderTest	187
BackTrackMem	250
BackTrackMemTest	242
PuncturingScheduler	41
PuncturingSchedulerTest	62
Salsa1	243
Salsa2	216
SalsaTest	128
Sorter	162
SorterTest	402
SpineEvaluator	299
SpineEvaluatorTest	114

Table 5: Source code analysis

Table 5 gives a summary of the source code required in this project. The total lines of source code was 3104. Of these, the total lines of test code was 1135 and non-test code was 1969. Thus roughly one third of our source code was testbench source code that did not have any bearing on the final FPGA implementation. This reinforces the notion of Bluespec being a highly test-driven development platform and is indicative of the amount of time we spent writing the code as compared against the amount of time we spent testing it.

5 Design Space Exploration

5.1 Improving Hash Module

We note that each *doubleround* operation consists of a *columnround* followed by a *rowround*. Each in turn consists of four independent *quarterround* operations (where we understand independent to mean temporal independence with respect to assignment). However, we note that *quarterround* operations are *not* independent, since each of y_0, y_3, y_2 according to (2) depends on the given assignment of y_3, y_2, y_1, respectively. In software this is not a problem, in-fact we are so used to procedural assignment that it would be very easy not to notice this complication. However, in hardware this presents us with several options if we are to achieve correct behavior. Either we must break down the *quarterround* operation into 4 separate operations and apply them successively, or we must explicitly express the *quarterround* operation into a combinational one that does present temporal dependencies. In the former case, this means that the Salsa hash function no longer takes 12 cycles, but $12 \times 4 \times 2 = 48$ cycles (with each *doubleround* operation implemented as 2 4-cycle *quarterround* operations). In the latter case, this means an increase in area, since we must explicitly short-circuit the assignment dependencies by adding combinational logic. We will refer to the former case as Salsa1 and the latter as Salsa2. Both versions were implemented and studied in this project.

5.2 Speed, Latency, Throughput

Table 6 shows the timing summary as part of the FPGA synthesizing results. We notice that the frequency of the decoder is not influenced by the structure of Salsa modules, which indicates that the frequency bottleneck of this is not in the Salsa module. This is due to the fact that the running frequency of a module is determined by the largest combinational path of the circuit, and this bottleneck exists elsewhere, possibly in the Sorter or Backtrack Memory modules.

	Salsa0	Salsa1	Salsa2
Minimum period:	10.200ns	10.200ns	10.200ns
Maximum Frequency:	98.035MHz	98.035MHz	98.035
Minimum input arrival time before clock:	3.528ns	3.528ns	3.528ns
Maximum output required time after clock:	6.984ns	6.984ns	6.984ns
Maximum combinational path delay:	1.170ns	1.170ns	1.170ns

Table 6: Timing Summary of Decoder

We also evaluated the speed bottleneck of our entire system. We added counters at the beginning and ending of each sub-module to test the total time each sub-module takes, and we list the results of the timing analysis in Table 7. We can see that the Salsa module and the Sorter module have the most cycles. In order to speed up this

decoder, we will need to optimize the structure of these two modules. The way of optimizing the Salsa module is discussed in Section 5.1.

Name of Actions	cycles with Salsa0	cycles with Salsa1	cycles with Salsa2
Salsa	1	51	15
SpineEvaluator	36	1663	516
doACS	1	1	1
Suggupd	1	1	1
evalupd	1	1	1
Searcher	34	34	34
Sorter	149	149	149
BacktrackMem	1	1	1
Decoder	12881	136583	49170

Table 7: Cycle Counting of Modules

Based on the number of cycles each module takes, we are able to calculate the latency and throughput of the entire decoder. In this report, we define latency as the time which the decoder needs to process the entire message. And throughput is defined as the bit rate at which the decoder can run when constantly fed with messages. Based on this, since the synthesized FPGA frequency is 98.035MHz, we can calculate the latency of the Salsa0, Salsa1 and Salsa2 versions of the decoder by:

$$t_{salsa0} = \frac{12881 cycles}{98.035 MHz} = 0.00013(seconds)$$
$$t_{salsa1} = \frac{136583 cycles}{98.035 MHz} = 0.00139(seconds)$$
$$t_{salsa2} = \frac{49170 cycles}{98.035 MHz} = 0.00050(seconds)$$

And we can calculate the throughput of the decoder by:

$$r_{salsa0} = \frac{192 bits}{0.00013 seconds} = 1.477(Mbits/s)$$
$$r_{salsa1} = \frac{192 bits}{0.00139 seconds} = 0.137(Mbits/s)$$
$$r_{salsa2} = \frac{192 bits}{0.00050 seconds} = 0.384(Mbits/s)$$

Table 8 summarizes the bit rate, latency and throughput of this decoder implementation.

	Salsa 0	Salsa 1	Salsa 2
Throughput	1.477 Mbits/s	0.137 Mbits/s	0.384 Mbits/s
Latency	0.00013 s	0.00139 s	0.00050 s
Frequency	98.035 MHz	98.035 MHz	98.035 MHz

Table 8: Decoder bit rate, latency, throughput

5.3 How much better can we do?

The results outlined in Section 5.2 suggest that as it stands the decoder implementation is not practical in a real system. However, we argue that with several feasible modifications in future work we can achieve an increase in performance of around 2 orders of magnitude, thus making the multi-pass decoder practical in a real communications system.

For this discussion we will use the results of the Salsa0 decoder, i.e. first, we assume that a hash module exists that performs hashing in 1 cycle.

Second, for the Sorter module we used a naive $O(n^2)$ algorithm. However, we are only interested in the best B suggested `MarkedCost`. As a result, the complexity of our sorting algorithm is in the order of $O(4n)$. By searching literature we find that there exist several potential algorithms that can reduce the complexity. For example, Timsort and Bubble Sort algorithms will be able to reach a best case complexity of $O(n)$. Currently the Sorter takes 149 cycles to complete. If an $O(n)$ algorithm is implemented, we reduce this to 32 cycles (for the same parameters). This increases our bit rate by a factor of approximately 5, giving us an improved bit rate of $\approx 1.5 \times 5 = 7.5$ Mbits/s.

Third, in the Spine Evaluator we note that a single hash module was used to handle all $B \times 2^k$ hashing requests sequentially. Current area usage indicates that for the given parameters we can modify the architecture to use B hashing modules in parallel, i.e. a separate hashing module is used for each spine. This increases our bit rate by a factor of 4 (for the given parameters), giving us an improved bit rate of $\approx 7.5 \times 4 = 30$ Mbits/s.

Finally, we may reasonably consider a Spine Evaluator that uses $B \times 2^k$ separate hashing modules. This is infeasible given the current space requirements of Salsa, but assuming that a more suitable hashing module is implemented, this increases our bit rate by a factor of 32 (for the given parameters), giving us an improved bit rate of $\approx 7.5 \times 32 = 240$ Mbits/s.

Of-course, this is a rough estimate of the increase in performance. But we can see that with three feasible modifications we can achieve roughly two orders of magnitude increase in performance. This suggests that the multiple pass decoder will indeed be practical.

5.4 Concurrency and Pipelining

As already described the decoder has been implemented such that it happens in 4 stages: Enumeration stage (`doEnumerate`), Add-Compare-Select Stage (`doACS`) stage, Suggestion updates (`suggupd`) stage and Spine Evaluator update (`evalupd`) stage. The decoder uses a register called `stage` which keeps track of an enum type called `Stage` which is used to make sure that the 4 stages occur in a sequential order. To improve on this implementation, it would be nice to make the stages run in parallel and hence get rid of the `stage` register.

The first step was realizing that the Spine Evaluator update stage was not necessary. The way the algorithm is designed, the Spine Evaluator calculates $B \times 2^k$ symbols for the ACS stage to compute costs. Yet the practical decoder keeps track of the best B suggestions from the previous stage so somehow the Spine Evaluator needs to be updated with the best B hashes from the previous code step. Even though this was previously done after backtrack memory updates itself, it is worth noting that all the information necessary for the updates is available the moment the sorter is done. Hence instead of sending the updates in an explicit Spine Evaluator update stage, that stage was eliminated and the updates were sent within the suggestion updates stage. Consequently the Spine Evaluator can update itself while backtrack memory is updating itself. This brings the decoder down to have only 3 stages instead of 4.

Another thing to consider too are the registers which are being used and updated within the stages: the suggestion costs register (`curr_suggcosts`) and the current schedule register (`curr_schedule`). We see that the puncturing scheduler continuously updates `curr_schedule` when it is called, while other stages read that register and react accordingly to the scheduler. Since the Spine Evaluator takes a relatively long time, the Enumerate stage will fire multiple times before the ACS stage receives spine symbols for the first code step. Hence instead of having the ACS stage read the schedule form the `curr_schedule` register, a FIFO queue of schedules could be added between the Enumeration stage and the ACS stage to make sure that the ACS stage does not react to the wrong schedule and the schedules and symbols happen in order. A FIFO queue of schedules could be added between the ACS and suggestion update stage for the same reason: so that the backtrack memory gets updates and is not mistaken with the wrong schedule. In order for the FIFOs to support `enq` and `deq` at the same time, they need to be either bypass FIFOs or two- or more element FIFOs.

As for the suggestion cost registers, the costs are accumulated as the decoder is running and hence it is important to make sure that they are updated accordingly. We notice that ACS reads this register while the suggestion update stage writes to this register. Hence semantically `doACS` $>$ `suggupd` according to the concurrency properties of the `curr_suggcosts` register. Yet these semantics make the algorithm become incorrect because we always want `suggupd` $>$ `doACS` so that `curr_suggcosts` is updated accordingly. Consequently we can create a special register module which wraps an actual register and an `RWire`. Every time a write is done to this register, the new value will be bypassed to ACS instead of the stale value which is in the wrapped register. Hence to make this work `curr_suggcosts` stage should be kept in a special register module.

With this it is possible to have the 3 stages run in parallel. Of course, the bottlenecks of the Sorter and the Spine Evaluator will still limit the speed of the decoder.

6 Future Work

This project provides a solid and fully parameterizable backbone for a Cortex decoder. This work can be extended with a view to deploying the decoder on the airwaves, and the authors sincerely hope that it proves useful towards this goal. Before that is possible, there are several challenges remaining that proved beyond the scope of this project (the most important of these have already been outlined in Sections 5.3 - 5.4).

First, the decoder needs to be generalized to multiple passes. The current implementation of the backtrack memory is not extensible to multiple passes, and does not store a complete record of all encountered symbols. Meeting this requirement will almost certainly require a separate memory module and a redesign of the backtrack memory architecture.

Second, the sorting algorithm needs to be reconsidered and a better one employed that will be able to guarantee an average case complexity of at least $O(n \log n)$ and a best case complexity of $O(n)$.

Third, the use of Salsa as a practical hashing mechanism should be re-evaluated and a more suitable substitute should be found. If Salsa is still used, then it should be extended to generate output symbols beyond 512 bits which is quite a simple modification. Furthermore, the Spine Evaluator should make use of multiple hashing modules in parallel to achieve maximum bit rate.

Finally, provisions should be made for the feedback protocol between the encoder and decoder. With this in place, the decoder can be considered for porting to a practical communications platform such as Airblue [2].

References

[1] Daniel J. Bernstein. Salsa20 specification. Technical report, The Univesity of Illinois at Chicago, 2005.

[2] Man Cheuk (Alfred) Ng, Kermin Elliott Fleming, Mythili Vutukuru, Samuel Gross, Arvind, and Hari Balakrishnan. Airblue: A System for Cross-Layer Wireless Protocol Development. In *ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, La Jolla, CA, October 2010.

[3] Jonathan Perry, Hari Balakrishnan, and Devarat Shah. Rateless wireless networking with spinal codes, January 2011.