FPGA Design and Implementation of the Joint Viterbi Detector Decoder

Brahim HAMADICHAREF and Kheong Sann CHAN
Non-Volatile Memory (NVM), Data Storage Institute (DSI)
Agency for Science Technology and Research (A*STAR)
2 Fusionopolis Way #08-01 Innovis, Singapore 138634

***Abstract*—In this paper we present the design and implementation of the Joint Viterbi Detector Decoder (JVDD) algorithm onto a Field Programmable Gate Array (FPGA). Based on the reference C-code, we implemented each individual functions of the JVDD in VHSIC Hardware Description Language (VHDL) and tested against the results of the reference C-code. They include the calculation of the noise free metrics, of the branch metrics, the copy-and-extend of the survivors' bit-patterns, survivors' kill-by-ParityCheck, kill-by-Threshold and kill-by-Capping, as well as sorting. Some modules exploit the full hardware parallelism, while trying to automate the generation of VHDL modules together with their test-benches. Our JVDD implementation can be parameterized for any Parity Check Matrix (PCM) size and code rate (R), with the current focus for a CWL = 64 and R = 0.5, scaling up to larger CWLs using Block RAMs (BRAMs). Finally, a systematic study of the JVDD memory requirements for longer CWLs (e.g. 4096) identified the need for larger FPGA, such as Xilinx Virtex-7 UltraScale.**

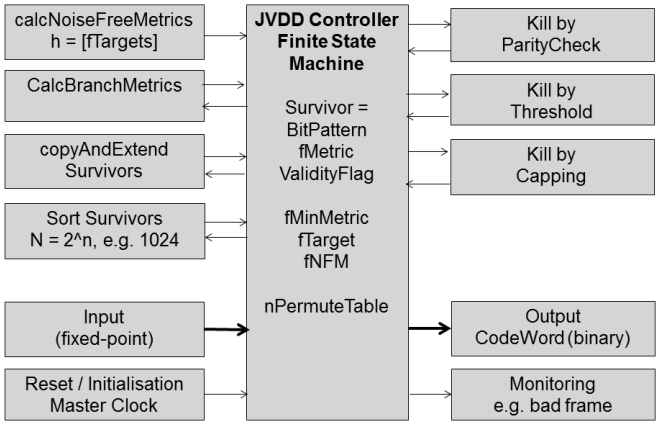

Fig. 1. FPGA Architecture of the Joint Viterbi Detector Decoder

I. Introduction

The Joint Viterbi Detector Decoder (JVDD) has been recently proposed as an alternative to the low density parity check (LDPC) decoder [1][2]. It is based on two important algorithms: the Viterbi Algorithm (VA) [3] and syndrome checking [4]. One important aspect of the JVDD is that, unlike the LDPC decoder, it is not iterative. More detail on the JVDD can be found from the original publication [5]. The C-code implementation of the JVDD, running on a large number of multi-cores Linux servers, has been used to evaluate various key aspects of the decoder including the development of codes [6][7].

In order to fully evaluate the performance of the JVDD, it is necessary to assess its performance in both the waterfall region, as well as the error floor region [8], corresponding to the region where the error rate vs SNR drops more slowly at high SNRs. Error floor evaluation is an important and challenging aspect of the design of the LDPC code [9] and applies equally to the design of the JVDD and its codes. This is part of the motivation for going to a hardware implementation of the JVDD.

In this paper, we present our progress on the design and implementation of the JVDD in hardware, with Field Programmable Gate Array (FPGA) as the target platform, and discuss challenges we face.

The rest of this paper is organized as follows. In Section II we present the JVDD hardware architecture and detail all its associated modules. In Section III we discuss important aspects of our work namely the number of survivors and performance metric, together with the evaluation framework. Finally, in Section IV we conclude the paper.

II. Hardware Architecture

A. Architecture

The overall architecture of the FPGA JVDD implementation is shown in Figure 1. At the heart of the JVDD is a controller implemented as a Finite State Machine (FSM). The controller orchestrates the flow of data, from the input to the different modules in sequence, to deliver an output. Both *Initialization* and *calcNoiseFreeMetrics* modules are called once after the reset state. The controller then goes through the different states triggering the modules in sequence: *Input*, *calcBranchMetrics*, *Copy-And-Extend*, *Kill-by-ParityCheck*, *Kill-by-Threshold*, *Kill-by-Capping*, *Sorting Survivors*, repeatedly as the decoder is fed with inputs. The *Output* module gives the final valid codeword output.

B. calcNoiseFreeMetrics

At the initialization stage, the NoiseFreeMetrics values are calculated from the target values as the noise-free output of the target response. In simulations, we assume a short 3-tap target of h=[0.408, 0.816, 0.408]. These NoiseFreeMetrics values are used, subsequently, in the calculation of the BranchMetrics values in the trellis [3]. The resulting hardware circuit implementing the calcNoiseFreeMetrics function is show in Figure 2

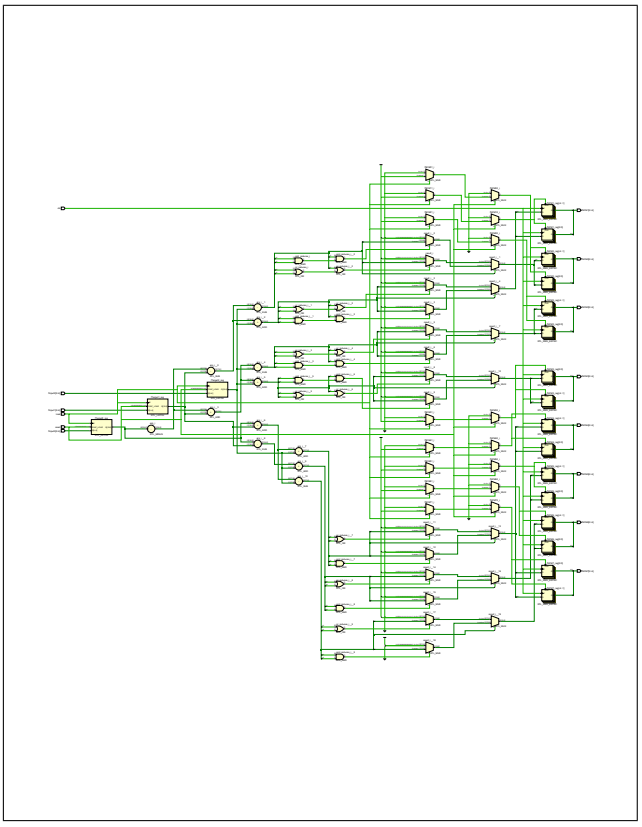

Fig. 2. Hardware circuit for the calcNoiseFreeMetrics function

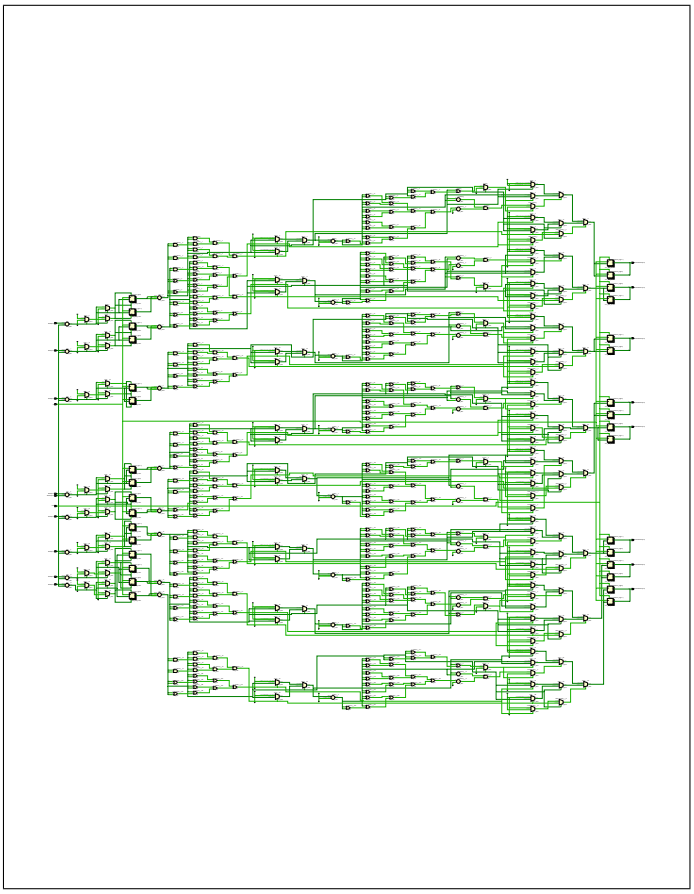

Fig. 3. Hardware circuit for the calcBranchMetric function

synthesized by Vivado Design Suite [10]. The 3 inputs (3-tap target h vector) follow path of adders and substracters to produce the 8 NoiseFreeMetrics outputs. Data path are fixed-point quantized in format Q(8.8) (8-bits for range and 8-bits for precision) [11]. Note that the reader can zoom in Figure 2 to see details of the hardware circuit.

C. calcBranchMetrics

Every input fed to the JVDD requires the calculation of the branch metrics which are summed together to form the path metrics that determine how close the particular path is to the received waveform. The NoiseFreeMetrics are used to calculate the branch metric for each state transitions (00→00, 00→10, 01→00, 01→10, 10→01, 10→11, 11→01 and 11→11). The resulting hardware circuit design for calcBranchMetrics is rather complex, as shown in Figure 3. For each of the 8 BranchMetrics values the hardware circuits square the difference between each NoiseFreeMetrics and the input reader value.

D. Copy-And-Extend

For each input value fed to the JVDD, the number of survivors increases using the Copy-And-Extend module, splitting each valid survivor into two new survivors, one with a bit '1' and one with a bit '0' appended to the survivor's bitPattern. This process grows the list of potential survivors, each time doubling its size. Unless some of the survivors get killed, their growth is not sustainable from a memory resources point of view. Two mechanisms, *Kill-by-ParityCheck* and *Kill-by-Threshold*, are used to limit the number of survivors as detailed in the following subsections.

E. Kill-by-ParityCheck

The survivors accumulated by the JVDD must be valid codewords and thus pass through every parity check, as they progress through the trellis structure [5][12]. Parity check modules are created from the Parity Check Matrix (PCM), from which the generator matrix is subsequently computed, normally as a dense matrix (See Figure 4). The size of PCM ($M \times N$ where $M = N - K$, K is the number of data bits and N is the number of coded bits) depends on the Code Word Length (CWL) and Code Rate ($R = K/N$).

MATLAB [13] scripts were developed to automatically generate the parity check circuits, in VHDL [14], for both the JVDD encoder based on the generator matrix and the decoder based on the PCM. All the *Kill-by-ParityCheck* circuits can be working in partial or full parallelism, exploiting hardware parallelism for performance and thus decoder throughput. We use the generator matrix (e.g. CWL=64 and R=0.5), as shown in Figure 4, to design the hardware circuit implementing the JVDD encoder. As shown in Figure 5, the encoder module

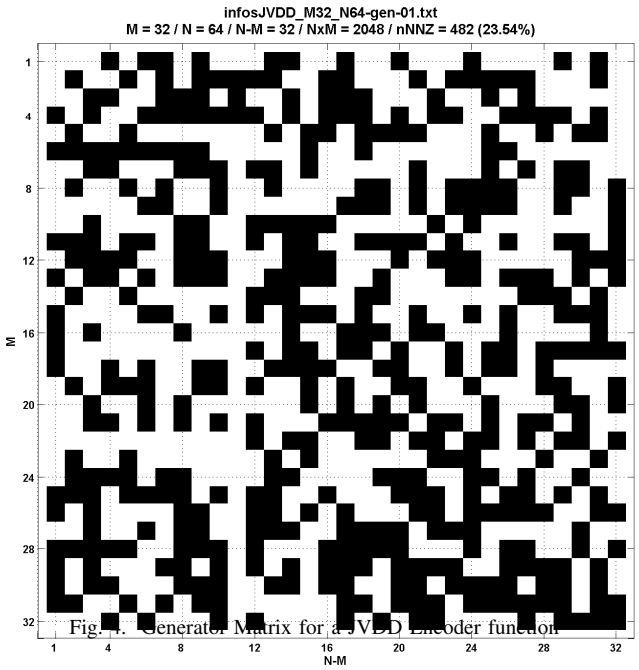

Fig. 4. Generator Matrix for a JVDD Encoder function

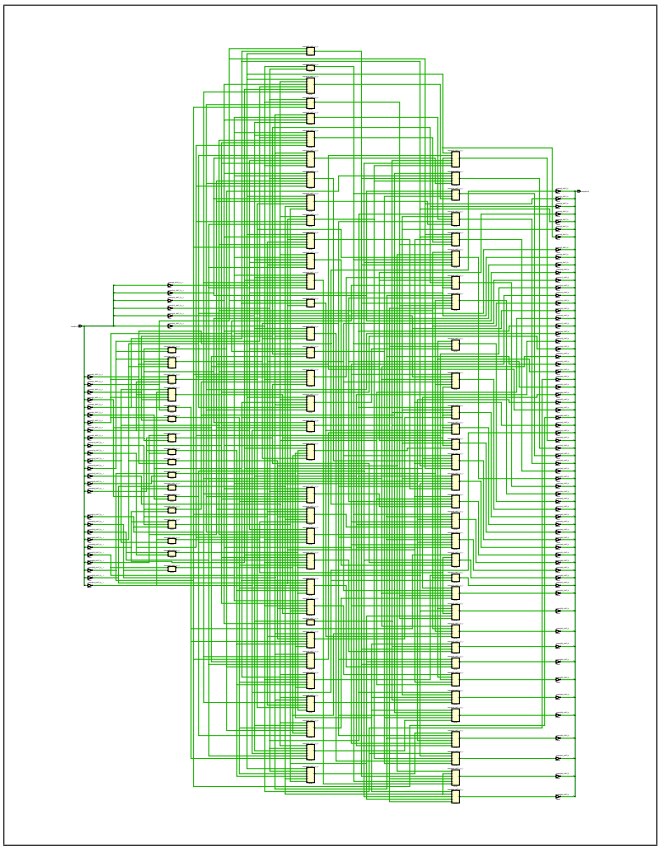

Fig. 5. Hardware circuit implementing the JVDD Encoder function

requires a large number of Look-Up Tables (LUT) [15] used to optimize the parity calculation.

F. Kill-by-Threshold

To progressively reduce the number of survivors, a *Kill-by-Threshold* module is used to kill survivors with metric larger than the current minimal metric plus a user-selected threshold (typically 1.0, 1.5, etc.). The smaller the threshold, the more survivors are potentially killed and the higher the probability of losing the Minimum Metric Legal Codeword (MMLC). However when the threshold is kept large the number of survivors will accumulate, leading to more memory resources and more complex management. The threshold must be chosen to appropropriately trade-off performance against complexity, from our experiments, a good number of survivors should be around four times CWL to help JVDD achieve good performance.

G. Kill-by-Capping

In the reference C-code implementation, the JVDD can easily use thousands of survivors, especially at longer CWL. In hardware however, memory is limited and costly, whether implemented in Distributed RAM or Block RAM [16]. Thus a *Kill-by-Capping* module is used to limit the number of survivors which would have not been killed by parity check or thresholding functions. This module is often not necessary for short CWL.

H. Sorting Survivors

The JVDD can be considered as a List Viterbi (LV) [17] type of algorithm, keeping a constrained number of potential survivors, and aiming to keep the best candidate as its final output. Thus the management of these survivors requires sorting to consistently keep track of the survivor with the smallest metric and to choose the largest metric survivors to discard during capping. We chose the odd-even merge as the best candidate implementation [18][19][20]. Newer JVDD improvements are being developed to remove the need for sorting survivors aiming to reduce hardware resources and improve the throughput of JVDD.

I. JVDD Output

Similar to LDPC decoders which may return an error when reaching the total number of iterations without success, the JVDD output, due to its design, may fail to return the MMLC. The process of *Copy-And-Extend* doubles the number of survivors, but *Kill-by-ParityCheck* and *Kill-by-Threshold* can potentially wipe out all the survivors. It is worth noting that *Kill-by-ParityCheck* is a necessary process as survivors must be valid codewords, while *Kill-by-Threshold* is used to help limit the number of survivors, low threshold value killing more survivors.

III. Discussion

A. Number of Survivors

In order to perform well the JVDD needs a large number of survivors. This is proportional to the code word length (CWL), i.e. larger CWL needs larger number of survivors to maintain good performance. We estimated the memory requirements of

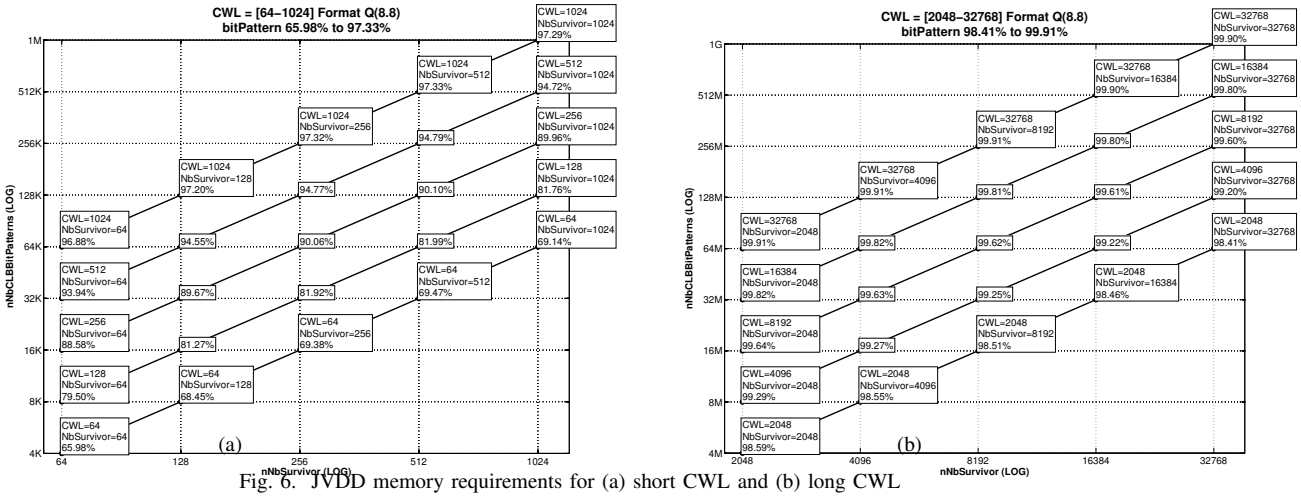

Fig. 6. JVDD memory requirements for (a) short CWL and (b) long CWL

JVDD, using Configurable Logic Block (CLB) [15] or Block RAM [16]. The bitPattern of each survivor uses the overall majority of the memory requirements, as it stores the final codeword (CWL bits) of the survivor. As shown in Figure 6a, the percentage of memory for bitPattern range from 43.84% (CWL = 32, NbSurvivors = 32) to 97.29% (CWL = 1024, NbSurvivors = 1024). At long CWL, as shown in Figure 6b, this percentage is well above 95%. Such memory requirements reach a 1Gbit, which is very high for currently available FPGA hardware [21]. For our future implementations with long CWL, we choose the Xilinx FPGA Virtex-7 UltraScale XCVU190-2FLGC2104E, featuring 132.9Mb of BRAM [22] as the Xilinx Virtex-7 UltraScale FPGA VCU110 development kit.

As shown in Figure 7 summarizing the JVDD decoder, in addition to the bitPattern, each survivor has other variables with a metric represented in fixed-point format Q(8.8), a state (2-bits if NbStates = 4, i.e. 00, 01, 10, 11) and a validity flag (1-bit used for representing if the survivor is still valid), however they take little memory space compared to bitPattern.

JVDD Controller
Finite State Machine

```
-- NbStates=4, NbH=3; CWL=64, NbSurvivors=256; N=8, M=8
-- Q(N.M) fixed-point unsigned signal with N bits for decimal part and M bits for fractional part

-- Survivors variables
type BitPatterns is array (NbSurvivor-1 downto 0) of std_logic_vector(CWL-1 downto 0);
type fMetrics is array (NbSurvivor-1 downto 0) of ufixed(N-1 downto -M);
type nState is array (NbSurvivor-1 downto 0) of std_logic_vector(1 downto 0); -- if NbStates=4
type bValidityFlags is array (NbSurvivor-1 downto 0) of std_logic;

-- Global variables
-- ISI FIR Filter [1 2 1] / sqrt(6)
fThreshold ufixed(N-1 downto -M);
type fTarget is array ((2^NbH)-1 downto 0) of sfixed(N downto -M);
type fMinMetric is array (NbState-1 downto 0) of ufixed(N-1 downto -M);
type fBranchMetric is array ((2^NbH)-1 downto 0) of ufixed(N-1 downto -M);
type fNFM is array ((2^NbState)-1 downto 0) of sfixed(N downto -M);
type nPermuteTable is array (NbSurvivor-1 downto 0) of integer range NbSurvivors-1 downto 0;
```

Fig. 7. Summary of the JVDD variables in VHDL format

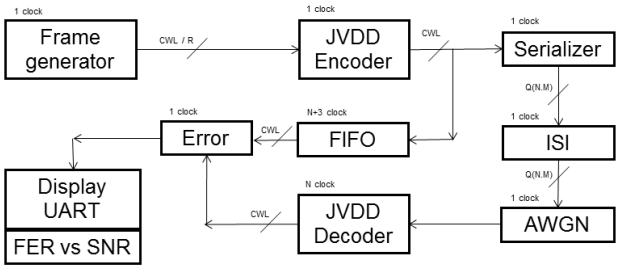

Fig. 8. Evaluation Framework for Communication Systems

B. Performance Metric

The performance metrics used to assess the JVDD are the classical Frame-Error-Rate (FER) and Bit-Error-Rate (BER) at various SNRs. Our motivations to implement the JVDD in FPGA are twofold. First it provides rapid prototype hardware version which ultimately will serve as the basis of an Application Specific Integrated Circuit (ASIC) for potential commercialization [12]. Second, it provides a mean to test the JVDD to assess error-floor level, in a significantly reduced time [23]. There is a need to run large Monte-Carlo (MC) simulations for quadrillions of frames to reach error-floor levels. For large CWL, e.g. 4096, reaching the error floor is not practical by software simulations.

C. Evaluation Framework

The JVDD hardware implementation is part of a FER evaluation framework as shown in Figure 8. FER evaluation systems are typically used to assess two important characteristic aspects of FER vs SNR curves: the waterfall and error-floor. For LDPC codes, these are usually attributed to trapping sets or near–codewords [24][8].

The error module calculates the Hamming distance between the original codeword frame (from the frame generator) and received codeword frame (output of the JVDD) and accumulates the number of frame errors and bit errors. The results are then converted to ASCII strings, using large binary to BCD converter [25], and finally sent to the console via the UART serial link. Results are archived and plotted using MATLAB.

IV. Conclusion

In this paper we presented our progress on the design and implementation of the JVDD on FPGA. The JVDD is implemented in separate VHDL modules, synthesized and implemented on Xilinx Virtex-7 FPGA using the Vivado Design Suite [10]. All modules are fully tested by exhaustive simulations with test-benches, and at all time, verifying their functionality and behavior against the reference C-code. The choice of the the JVDD architecture is very important as it dictates data flow and hardware resource requirements.

The current JVDD implementation at short CWL is being extended to larger CWL for 1024, 4096, etc. As LDPC is used in standard IEEE 802.3an (10GBase-T) Ethernet [26] and recently the Second Generation Standard for Satellite Broad-Band Services (DVB-S2) [27]. In order to meet the requirements of these applications, the JVDD will need to be modified and tested at very long CWLs approaching 64800 bits.

Acknowledgment

We acknowledge financial support from National Research Fund (NRF) Singapore (The Joint Viterbi Detector/Decoder (JVDD) for Satellite Communications, NRF2013SAS-SRP001047). The JVDD is protected under US Patent 9048987 [12].

References

[1] R. Gallager, "Low-density parity-check codes," *IEEE Transactions on Information Theory*, vol. 8, no. 1, pp. 21–28, 1962.

[2] T. J. Richardson, M. A. Shokrollahi, and R. L. Urbanke, "Design of Capacity–Approaching Irregular Low–Density Parity–Check Codes," *IEEE Transactions on Information Theory*, vol. 47, no. 2, pp. 619–637, February 2001.

[3] G. D. Forney, "The viterbi algorithm," *Proceedings of the IEEE*, vol. 61, no. 3, pp. 268–278, March 1973.

[4] S. Owada, "Parity check circuit," U.S. Patent 6027243, February 2000.

[5] K. S. Chan, S. S. B. Shafiee, E. M. Rachid, and Y. L. Guan, "Optimal Joint Viterbi Detector Decoder (JVDD) over AWGN/ISI channel," *Proceedings of the 2014 International Conference on Computing, Networking and Communications (ICNC2014), Hawaii, USA, February 3–6, 2014*, pp. 282–286.

[6] S. S. B. Shafiee, K. S. Chan, and Y. L. Guan, "A Performance Study of the Joint Viterbi Detector Decoder (JVDD) with GDLD and GDPD Codes," *Proceedings of the 2014 International Conference on Computational Intelligence and Communication Networks (CICN2014), Bhopal, India, November 14–16, 2014*, pp. 339–343.

[7] A. James, K. S. Chan, and S. Shafidah, "Sparse construction of joint Viterbi detector decoder (JVDD) codes," *Proceedings of the 11th Advanced International Conference on Telecommunications (AICT2015), Brussels, Belgium, June 21–26, 2015*, pp. 29–33.

[8] Y. He, J. Yang, and J. Song, "A survey of error floor of LDPC codes," *Proceedigns of the 2011 6th International ICST Conference on Communications and Networking in China (CHINACOM2011), Harbin, China, August 17–19, 2011*, pp. 61–64.

[9] S. Zhang and C. Schlegel, "Controlling the Error Floor in LDPC Decoding," *IEEE Transactions on Communications*, vol. 61, no. 9, pp. 3566–3575, 2013.

[10] Xilinx, "Vivado Design Suite – HLx Editions," *Productivity. Multiplied.* [Online]. Available: http://www.xilinx.com/products/design-tools/vivado.html

[11] A. Gersho, "Principles of quantization," *IEEE Transactions on Circuits and Systems*, vol. 25, no. 7, pp. 427–436, July 1978.

[12] K. S. Chan, M. R. Elidrissi, and S. S. B. Shafiee, "Joint detector/decoder devices and joint detection/decoding methods," U.S. Patent 9048987, June 2015.

[13] MATLAB – The Language of Technical Computing, *Online Documentation*, The MathWorks Inc., 2016, http://www.mathworks.com/help/matlab/.

[14] IEEE Computer Society, "IEEE Standard VHDL Language Reference Manual," *IEEE Std 1076-2008 (Revision of IEEE Std 1076-2002)*, pp. 1–620, 2009.

[15] Xilinx, "7 Series FPGAs Configurable Logic Block – User Guide," *UG474 (v1.7) November 17, 2014*.

[16] ——, "7 Series FPGAs Memory Resources – User Guide," *(UG473 (v1.11) November 12, 2014)*. [Online]. Available: http://www.xilinx.com/support/documentation/user_guides/ug473_7Series_Memory_Resources.pdf

[17] N. Seshadri and C.-E. W. Sundberg, "List Viterbi decoding algorithms with applications," *IEEE Transactions on Communications*, vol. 42, no. 234, pp. 313–323, Feb/Mar/Apr 1994.

[18] A. Farmahini–Farahani, H. J. Duwe III, M. J. Schulte, and K. Compton, "Modular Design of High–Throughput, Low–Latency Sorting Units," *IEEE Transactions on Computers*, vol. 62, no. 7, pp. 1389–1402, July 2013.

[19] K. E. Batcher, "Sorting networks and their applications," *Proceedings of the American Federation of Information Processing Societies (AFIPS'68), April 30–May 2, 1968*, pp. 307–314.

[20] C. D. Thompson, "VLSI Complexity of Sorting," *IEEE Transactions on Computers*, vol. C-32, no. 12, pp. 1171–1184, 1983.

[21] Xilinx, "UltraScale Architecture and Product Overview," *DS890 (v2.7) February 17, 2016*. [Online]. Available: http://www.xilinx.com/support/documentation/data_sheets/ds890-ultrascale-overview.pdf

[22] ——, "Virtex UltraScale FPGAs Data Sheet: DC and AC Switching Characteristics," *DS893 (v1.7.1) April 4, 2016*.

[23] Y. Cai, S. Jeon, K. Mai, and B. V. K. V. Kumar, "Highly Parallel FPGA Emulation for LDPC Error Floor Characterization in Perpendicular Magnetic Recording Channel," *IEEE Transactions on Magnetics*, vol. 45, no. 10, pp. 3761–3764, 2009.

[24] T. Richardson, "Error floors of LDPC codes," *Proceedigns of the 41st Annual Allerton Conference on Communication, Control, and Computing, Urbana–Champaign, USA, October, 2003*, pp. 1426–1435.

[25] M. Benedek, "Developing Large Binary to BCD Conversion Structures," *IEEE Transactions on Computers*, vol. C–26, no. 7, pp. 688–700, July 1977.

[26] A. E. Cohen and K. K. Parhi, "A Low-Complexity Hybrid LDPC Code Encoder for IEEE 802.3an (10GBase-T) Ethernet," *IEEE Transactions on Signal Processing*, vol. 57, no. 10, pp. 4085–4094, October 2009.

[27] A. Morello and V. Mignone, "DVB-S2: The Second Generation Standard for Satellite Broad–Band Services," *Proceedings of the IEEE*, vol. 94, no. 1, pp. 210–227, January 2006.