FITEE

A multistandard and resource-efficient Viterbi decoder for a multimode communication system[*]

Yi-qi XIE[1,2], Zhi-guo YU[†‡1,2], Yang FENG[1,2], Lin-na ZHAO[1,2], Xiao-feng GU[1,2]

[1]MOE Engineering Research Center of IoT Technology Applications, Wuxi 214122, China

[2]Department of Electronic Engineering, Jiangnan University, Wuxi 214122, China

[†]E-mail: yuzhiguo@jiangnan.edu.cn

Abstract: We present a novel standard convolutional symbols generator (SCSG) block for a multi-parameter reconfigurable Viterbi decoder to optimize resource consumption and adaption of multiple parameters. The SCSG block generates all the states and calculates all the possible standard convolutional symbols corresponding to the states using an iterative approach. The architecture of the Viterbi decoder based on the SCSG reduces resource consumption for recalculating the branch metrics and rearranging the correspondence between branch metrics and transition paths. The proposed architecture supports constraint lengths from 3 to 9, code rates of 1/2, 1/3, and 1/4, and fully optional polynomials. The proposed Viterbi decoder has been implemented on the Xilinx XC7VX485T device with a high throughput of about 200 Mbps and a low resource consumption of 162k logic gates.

Key words: Reconfigurable Viterbi decoder; Multi-parameter; Low resource consumption; Standard convolutional symbols generator (SCSG); Fully optional polynomials

https://doi.org/10.1631/FITEE.1601596   **CLC number:** TN764

1 Introduction

In today's wireless communication systems, convolutional encoding coupled with the Viterbi algorithm has been widely used for forward error correction (FEC) (Chang et al., 2010; Kim et al., 2012; Yoo et al., 2012). With the diversification of the application scenarios, terminal devices are required to operate on multiple communication standards, and the Viterbi algorithm must support multiple standards. A number of studies have been devoted to

reconfigurable Viterbi algorithm in recent years (Swaminathan et al., 2002; Benaissa and Zhu, 2003; Niktash et al., 2006; Bissi et al., 2008; Vennila et al., 2013), mainly concentrating on dynamic reconfigurability, high throughput, and multiple parameters. Cavallaro and Vaya (2003) presented a reconfigurable Viterbi decoder with a flexible configuration, a constraint length of 3–9, a code rate of 1/2–1/3, and a throughput of 60 Mbps. Batcha and Sha'ameri (2007) implemented a reconfigurable Viterbi decoder with a high throughput of 150 Mbps. Campos and Cumplido (2006) suggested a dynamic partial reconfigurable Viterbi decoder. These reconfigurable Viterbi decoders can adapt to different applications. However, little research has been devoted to fulfilling all of the conditions for a reconfigurable Viterbi decoder, such as reconfigurable properties, speed, and resource consumption.

Resource consumption of a reconfigurable Viterbi algorithm lies mainly in the computation of standard convolutional symbols and the arrangement

---

of correspondence between branch metrics and transition paths. The speed and reconfigurable properties of the Viterbi decoder are determined by how many parallel schemes are implemented. To achieve the goals of multiple parameters and high speed with low resource consumption, we present a new block, the standard convolutional symbols generator (SCSG), in the reconfigurable Viterbi decoder. The SCSG block uses an iterative approach to generate all the states and the corresponding standard convolutional symbols, and then distributes all the standard convolutional symbols to the branch metric (BM) block to compute the branch metrics. The SCSG-based technique will form a kind of self-alignment between the branch metrics and the transition paths. The proposed reconfigurable Viterbi decoder based on SCSG has been verified, and the results show that it can support a constraint length of 3–9, code rates of 1/2, 1/3, 1/4, and a throughput of 200 Mbps with a low logic gate utilization of 162k.

2 The proposed reconfigurable Viterbi decoder architecture

Generally, a fixed Viterbi decoder consists of four major functional blocks: branch metric, add-compare-select (ACS), survivor path memory (SPM), and traceback. The BM block takes the input symbols and generates the branch metrics by computing the Hamming or Euclidean distance between the input symbols and standard convolutional symbols. The

ACS block computes the state metrics and survivor paths, and then sends the survivor paths to the SPM block. Finally, the traceback block rebuilds the decoded sequences according to survivor paths.

To accommodate multiple communication standards and low resource consumption, we suggest a new block, the SCSG. Fig. 1 shows the architecture of the proposed reconfigurable Viterbi decoder. The principle of the proposed decoder is described as follows:

1. The polynomials and the input symbols are normalized in the configuration unit (CU) block. Then the normalized polynomials are fed into the SCSG block to calculate the standard convolutional symbols.

2. The hard-to-soft unit in the hard-soft branch metric (HBM) block quantifies the standard convolutional symbols from two-level symbols to eight-level symbols, and each branch metric unit (BMU) in the HBM block computes the Euclidean distance between the normalized symbols and quantified standard convolutional symbols.

3. The ACS block computes the state metric by accumulating the branch metrics, and sends the most significant bit (MSB) of the previous states to the SPM block as the survivor paths. After that, the pipeline comparator in the ACS block compares all the state metrics of the current states and chooses the minimum one as the shortest path.

4. The traceback block traces back through the trellis according to the shortest path and survivor paths, and outputs the decoded sequences.
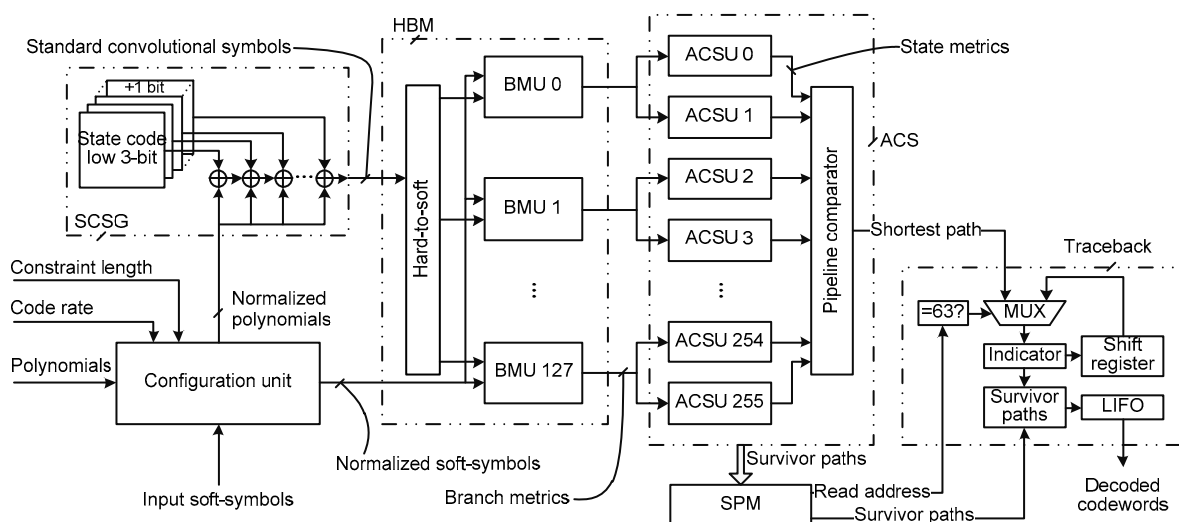

Fig. 1 Block diagram of the reconfigurable Viterbi decoder

2.1 Configuration unit

For the reconfigurable Viterbi decoder, the variation of the code rate with the constraint length can cause changes in the polynomials and input symbols. When the code rate is $1/R$ ($2 \leq R \leq 4$), the number of encoded symbols and polynomials from the encoder is R, and the Viterbi decoder is supposed to deal with the same number of input symbols and polynomials. When the constraint length is K ($2 \leq K \leq 4$), the bit widths of the polynomials in the encoder and the decoder are supposed to be K. The valid maximum number of input symbols and polynomials in our design is 4, the maximum width of each polynomial is 9, and each bit of the polynomials can be set optionally.

Fig. 2 illustrates the normalization of the polynomials and input symbols in the CU block. When the code rate and constraint length change, **Factor**$_0$, **Factor**$_1$, and **Factor**$_2$ are updated by the two mapping units, and then perform the AND operation with input symbols and original polynomials. Table 1 shows the mapping relationships between the normalized **Factor**$_0$, **Factor**$_1$, and the code rate, where each vector in **Factor**$_1$ corresponds to one polynomial. Table 2 shows the mapping relationship between the normalized **Factor**$_2$ and the constraint length.

To clearly describe the normalization procedure, let S and s be the input symbols and the normalized symbols, respectively, P_d, P_c, P_b, and P_a the 4×9 bit sequences of four original polynomials, and p_d, p_c, p_b, and p_a the four normalized polynomials. S performs a bitwise-AND operation with **Factor**$_0$. P_d, P_c, P_b, and P_a first perform bitwise-AND operations with the sequences in **Factor**$_1$, and then perform bitwise-AND operations with **Factor**$_2$. The normalization formulae can be deduced as

$$s = S \,\&\, \textbf{Factor}_0, \tag{1}$$

$$\begin{cases} p_d = (P_d \,\&\, \textbf{Factor}_2) \,\&\, \textbf{Factor}_1 <1>, \\ p_c = (P_c \,\&\, \textbf{Factor}_2) \,\&\, \textbf{Factor}_1 <2>, \\ p_b = (P_b \,\&\, \textbf{Factor}_2) \,\&\, \textbf{Factor}_1 <3>, \\ p_a = (P_a \,\&\, \textbf{Factor}_2) \,\&\, \textbf{Factor}_1 <4>, \end{cases} \tag{2}$$

where '&' is the bitwise-AND operation, **Factor**$_1$<1> indicates the first vector sequence in **Factor**$_1$, **Factor**$_1$<2> indicates the second vector sequence in **Factor**$_1$, and so on.
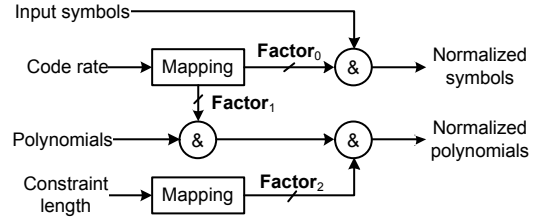

Fig. 2 Fabric of the configuration unit

Table 1 Relationships between Factors and code rate

Code rate ($1/R$)	Factor$_0$	Factor$_1$
1/2	0011	$\begin{bmatrix} 000000000 \\ 000000000 \\ 111111111 \\ 111111111 \end{bmatrix}$
1/3	0111	$\begin{bmatrix} 000000000 \\ 111111111 \\ 111111111 \\ 111111111 \end{bmatrix}$
1/4	1111	$\begin{bmatrix} 111111111 \\ 111111111 \\ 111111111 \\ 111111111 \end{bmatrix}$

Table 2 Relationships between Factors and constraint length

Constraint length (K)	Factor$_2$
3	111000000
4	111100000
5	111110000
6	111111000
7	111111100
8	111111110
9	111111111

2.2 Standard convolutional symbols generator

To create a reconfigurable Viterbi decoder supporting flexible polynomials, it is necessary to generate every possible standard convolutional symbol and recalculate the branch metrics corresponding to current states. In the SCSG block, we use an iterative method to generate the standard convolutional symbols of all even states, which range from 00000000 to 11111110. Because the only difference between even states and odd states is the last significant bit (LSB), the calculation of all odd states is

simple, and the computational process of odd states will be demonstrated in the final step. The SCSG first generates the starting standard convolutional symbols based on a series of 8-bit even states, 00000000, 00000010, 00000100, and 00000110, which are denoted as α. After that, the SCSG flips the LSB+3, LSB+4, LSB+5, LSB+6, and LSB+7 (MSB) of α in sequence to obtain the other even states, where we use LSB+i ($0 \leq i \leq 8$) to indicate the ith bit to LSB's left. The standard convolutional symbols of these even states can be formed by a few additional XOR operations with the starting standard convolutional symbols, which will greatly reduce resource consumption. After all the standard convolutional symbols of even states are generated, the standard convolutional symbols of all odd states can be formed by a few XOR operations with the LSB of the normalized polynomials. The procedure is organized in the following three steps:

1. SCSG calculating the standard convolutional symbols of α

Here we use $00000C_1C_00$ (C_1C_0 can be 00, 01, 10, or 11) to represent the state of α; the previous states corresponding to $00000C_1C_00$ in the trellis are $000000C_1C_0$ and $100000C_1C_0$. Let $\mathbf{SS}_N^0$ and $\mathbf{SS}_N^1$ be the starting standard convolutional symbol of the previous states $000000C_1C_0$ and $100000C_1C_0$, respectively. The SCSG first performs a bitwise-AND operation between the four normalized polynomials and $00000C_1C_00$, which results in $\mathbf{SS}_N^0$ by a reduction XOR on the results of bitwise-AND operations. $\mathbf{SS}_N^1$ can be obtained by an additional XOR operation between $\mathbf{SS}_N^0$ and the MSB of the normalized polynomials. The computational procedure of $\mathbf{SS}_N^0$ and $\mathbf{SS}_N^1$ is expressed as

$$\mathbf{SS}_N^0 = \{(\oplus(p_{d2}p_{d1} \& C_1C_0)), (\oplus(p_{c2}p_{c1} \& C_1C_0)), (\oplus(p_{b2}p_{b1} \& C_1C_0)), (\oplus(p_{a2}p_{a1} \& C_1C_0))\}, \quad (3)$$

$$\mathbf{SS}_N^1 = p_{d8}p_{c8}p_{b8}p_{a8} \wedge \mathbf{SS}_N^0, \quad (4)$$

where '$\oplus$' indicates the reduction XOR operation, '$\wedge$' indicates the bitwise-XOR operation, '$\{\}$' indicates the concatenation operation, '$\&$' indicates the bitwise-AND operation, p_{d2} indicates the LSB+2 of $\boldsymbol{p}_d$,

p_{c1} indicates the LSB+1 of $\boldsymbol{p}_c$, and so on.

2. SCSG calculating the standard convolutional symbols for all even states

The SCSG flips the LSB+3 of α to obtain the even state codes ranging from 00001000 to 00001110, which are denoted as β. The only difference between β and α is the LSB+3. Therefore, the standard convolutional symbols of β can be easily obtained from the starting standard convolutional symbols of α by an additional bitwise-XOR operation:

$$\mathbf{SS}_\beta = p_{d3}p_{c3}p_{b3}p_{a3} \wedge \mathbf{SS}_\alpha, \quad (5)$$

where $\mathbf{SS}_\alpha$ indicates the starting standard convolutional symbols $\mathbf{SS}_N^0$ and $\mathbf{SS}_N^1$, and $\mathbf{SS}_\beta$ indicates the standard convolutional symbols of β.

By the same token, the standard convolutional symbols for the rest of the even states from 00010000 to 11111110 can be deduced from Eqs. (6)–(9).

The even state codes from 00010000 to 00011110, which are denoted as δ, can be obtained by flipping the LSB+4 of α and β, and the standard convolutional symbols can be calculated by

$$\mathbf{SS}_\delta = p_{d4}p_{c4}p_{b4}p_{a4} \wedge \mathbf{SS}_{\alpha+\beta}, \quad (6)$$

where $\mathbf{SS}_{\alpha+\beta}$ indicates the standard convolutional symbols of α and β, and $\mathbf{SS}_\delta$ indicates the standard convolutional symbols of δ.

The even state codes from 00100000 to 00111110, which are denoted as ε, can be obtained by flipping the LSB+5 of α, β, and δ, and the standard convolutional symbols can be calculated by

$$\mathbf{SS}_\varepsilon = p_{d5}p_{c5}p_{b5}p_{a5} \wedge \mathbf{SS}_{\alpha+\beta+\delta}, \quad (7)$$

where $\mathbf{SS}_{\alpha+\beta+\delta}$ indicates the standard convolutional symbols of α, β, and δ, and $\mathbf{SS}_\varepsilon$ indicates the standard convolutional symbols of ε.

The even state codes from 01000000 to 01111110, which are denoted as η, can be obtained by flipping the LSB+6 of α, β, δ, and ε, and the standard convolutional symbols can be calculated by

$$\mathbf{SS}_\eta = p_{d6}p_{c6}p_{b6}p_{a6} \wedge \mathbf{SS}_{\alpha+\beta+\delta+\varepsilon}, \quad (8)$$

where $\mathbf{SS}_{\alpha+\beta+\delta+\varepsilon}$ indicates the standard convolutional symbols of α, β, δ, and ε, and $\mathbf{SS}_{\eta}$ indicates the standard convolutional symbols of η.

The even state codes from 10000000 to 11111110, which are denoted as μ, can be obtained by flipping the LSB+7 of α, β, δ, ε, and η, and the standard convolutional symbols can be calculated by

$$\mathbf{SS}_{\mu}=p_{d7}p_{c7}p_{b7}p_{a7}\wedge \mathbf{SS}_{\alpha+\beta+\delta+\varepsilon+\eta}, \tag{9}$$

where $\mathbf{SS}_{\alpha+\beta+\delta+\varepsilon+\eta}$ indicates the standard convolutional symbols of α, β, δ, ε, and η, and $\mathbf{SS}_{\mu}$ indicates the standard convolutional symbols of μ.

3. SCSG calculating the standard convolutional symbols for all odd states

To obtain the standard convolutional symbols for all odd states, the only computation needed is an additional bitwise-XOR operation with the LSB of p_{d}, p_{c}, p_{b}, and p_{a}:

$$\mathbf{SS}_{\tau}=p_{d0}p_{c0}p_{b0}p_{a0}\wedge \mathbf{SS}_{\alpha+\beta+\delta+\varepsilon+\eta+\mu}, \tag{10}$$

where $\mathbf{SS}_{\alpha+\beta+\delta+\varepsilon+\eta+\mu}$ indicates the standard convolutional symbols of the even states, and $\mathbf{SS}_{\tau}$ indicates the standard convolutional symbols of the odd states.

From Eqs. (3)–(10), we can see that, for all standard convolutional symbols except α, only a few XOR operations are needed to simplify the computations. Moreover, all the standard convolutional symbols correspond to their current states, which will help the HBM block match the branch metrics with the transition paths.

2.3 Hard-soft branch metric

The regular BM block in the fixed Viterbi decoder computes the branch metrics with fixed standard convolutional symbols, which cannot vary with the constraint length, code rate, or polynomials.

The HBM block consists of a hard-to-soft unit and several BMUs. Compared with the regular BM block, the HBM block computes the branch metrics with standard convolutional symbols from the SCSG block. When the polynomials, constraint length, and code rate change, the HBM block can recalculate the branch metrics, and the new branch metrics can self-align to each transition path.

The hard-to-soft unit converts the two-level

standard convolutional symbols to the eight-level ones. For example, if the standard convolutional symbol is 0_1_0_1, the eight-level quantified symbol is 000_111_000_111, which in decimal is 0_7_0_7.

Each BMU takes the normalized symbols and the quantified standard convolutional symbols, and computes the proximal Euclidean distance as the branch metrics (Xiong et al., 2004).

2.4 Add, compare, and select

The ACS block has the branch metrics as the inputs, and the survivor paths and the shortest path as the outputs. Every add-compare-select unit (ACSU) in the ACS block accumulates the branch metrics corresponding to current states, compares the sums, and saves the smaller sums as the state metrics of current states. The LSBs of previous states corresponding to current states are sent to the SPM block as the survivor paths. After all the state metrics of the current states are calculated, the results are fed into a pipeline comparator to compute the shortest path.

Because the standard convolutional symbols correspond one-to-one with the current states, the branch metrics self-align to the transition paths in the ACSU, simplifying the procedure of rearranging the routing between the branch metrics and current states. In Fig. 3, each group of BMUs takes the standard convolutional symbols from a group of current states, such as α and β, where '◎' indicates the bitwise-XOR operation with the LSB of the normalized polynomials, $p_{d0}p_{c0}p_{b0}p_{a0}$. The BMUs calculate the branch metrics, and then distribute the branch metrics to each group of ACSUs.
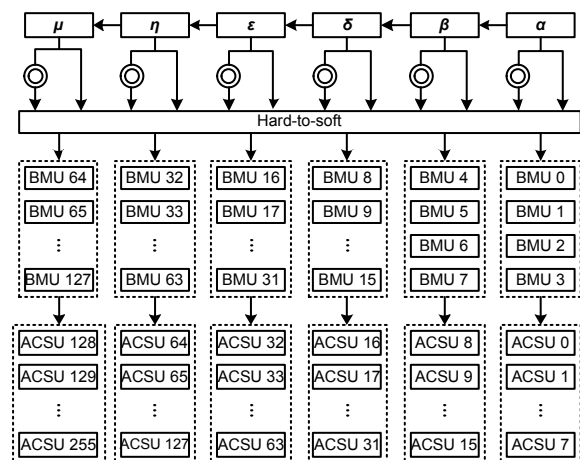

Fig. 3 Data flow of the BMUs and ACSUs

2.5 Survivor path memory

The SPM block is responsible for saving the survivor paths generated by the ACS block. To meet the truncation length of the reconfigurable Viterbi decoder, the size of the memory is set according to the largest constraint length and code rate decoder. Generally speaking, for a fixed Viterbi decoder, the truncation length is double or triple the value of $(K-1)\times[1\div(1-1/R)]$. Therefore, to meet the maximum constraint length 9 and the code rate 1/2, the optimum depth of the memory is 64, and the bit width is $2^{9-1}=256$.

2.6 Traceback

The traceback block consists of a traceback controller (TC) unit and a last-in-first-out (LIFO) unit. In Fig. 4, every time the read address of memory blocks in the SPM block rises to 63, the TC unit takes in the shortest path from the ACS block as the indexer, searches for the binary bits of the survivor paths according to the indexer, and then the decoded bit can be obtained. If the read address is not equal to 63, the value of the indexer can be obtained by shifting the previous value of the indexer and covering the LSB of the shifted indexer with the previous decoded bit. Finally, the LIFO unit reverses the order of decoded bits to obtain the decoded sequence.
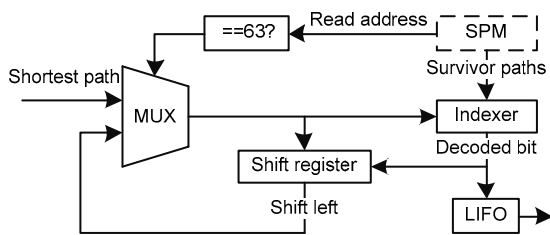

Fig. 4 Fabric of traceback

3 Implementation and results

To verify the bit error rate (BER) performance of the decoder, we implement the decoder for different parameters with the additive white Gaussian noise (AWGN) channel, and the BER curves are shown in Fig. 5. With the increasing constraint length and the decreasing code rate, BERs decline substantially, agreeing with the expected theoretical results (Moon, 2005). Moreover, we test the decoder with the same constraint length 9 and code rate 1/4 but different polynomials, and the results show that the variation in the polynomials results in different BER performance.
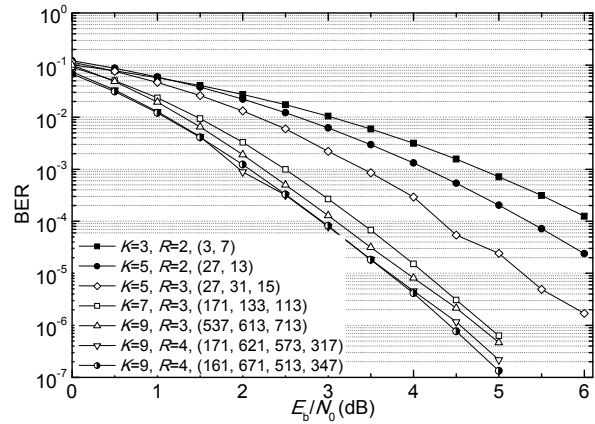

Fig. 5 Bit error rate of the proposed architecture
The numbers in parentheses indicate the polynomials in octal format

The proposed architecture has been implemented on a Xilinx Virtex-7 FPGA using Verilog hardware description language (HDL). The results show that the reconfigurable Viterbi decoder can support a constraint length of 3–9, code rates of 1/2, 1/3, 1/4, and fully optional polynomials. Moreover, the consumption of logic gates is limited to 162k. Fig. 6 shows the simulated waveforms for different code rates, constraint lengths, and polynomials. To facilitate the display of results, two 8-bit SerDes are added to the convolutional encoder and Viterbi decoder. One SerDes converts the 8-bit parallel sine wave to serial input symbols for the convolutional encoder, and the other one transforms the decoded sequences of the Viterbi decoder to an 8-bit parallel signal. When the communication standard changes, the signal 'reset' initializes the convolutional encoder and Viterbi decoder. After a period of time, the Viterbi decoder outputs the signals, which are the same as the input signals of the convolutional encoder.

4 Discussion

A comparison of the key features of our architecture with the works reported is shown in Table 3. Although some of the reported architectures were
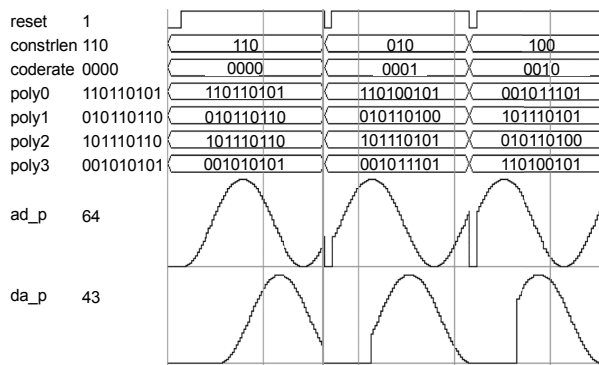
Fig. 6 Simulated waveforms

poly0, poly1, poly2, and poly3 represent four polynomials; ad_p indicates the input of convolutional encoder; da_p indicates the output of the Viterbi decoder; constrlen indicates the constraint length

implemented on different FPGA platforms, the authors gave the equivalent number of logic gates.

Compared with the reported reconfigurable Viterbi architecture, the proposed architecture can support an additional code rate of 1/4, which is commonly used in DAB and CDMA2000, and the throughput is higher than the others. The logic gate consumption of the SCSG-based reconfigurable Viterbi decoder is limited to 162k, showing 15% resource savings when compared with the architecture having the nearest performance (Cavallaro and Vaya, 2003).

Overall, the proposed architecture achieves flexible configuration and high throughput with a compact size, but there is still room for improvement. In future work, we will focus on the radix-4 technique and more flexible code rates, such as 2/3 and 3/4.

5 Conclusions

In this paper, we have presented a fully flexible reconfigurable Viterbi decoder with higher speed and smaller area. This architecture is an efficient decoder for a constraint length of 3–9 with code rates of 1/2, 1/3, and 1/4. The results showed that the SCSG-based architecture reduces resource consumption without reducing the flexibility or data rate. Such an architecture can support portable wireless devices in multimode communication.

References

Batcha MFN, Sha'ameri AZ, 2007. Configurable adaptive Viterbi decoder for GPRS, EDGE and Wimax. IEEE Int Conf on Telecommunications and Malaysia Int Conf on Communications, p.237-241.
https://doi.org/10.1109/ICTMICC.2007.4448640

Benaissa M, Zhu YQ, 2003. A novel high-speed configurable Viterbi decoder for broadband access. *EURASIP J Adv Signal Process*, 2003(13):1317-1327.
https://doi.org/10.1155/S1110865703310054

Bissi L, Placidi P, Baruffa G, et al., 2008. A Viterbi decoder architecture for a standard-agile and reprogrammable transceiver. *Integr VLSI J*, 41(2):161-170.
https://doi.org/10.1016/j.vlsi.2007.04.001

Campos JM, Cumplido R, 2006. A runtime reconfigurable architecture for Viterbi decoding. 3rd Int Conf on Electrical and Electronics Engineering, p.1-4.
https://doi.org/10.1109/ICEEE.2006.251908

Cavallaro JR, Vaya M, 2003. Viturbo: a reconfigurable architecture for Viterbi and Turbo decoding. IEEE Int Conf on Acoustics, Speech, and Signal Processing, p.497-500.
https://doi.org/10.1109/ICASSP.2003.1202412

Table 3 Comparison of key features of the architectures reported

Reference	Constraint length	Code rate	Maximum throughput (Mbps)	Number of logic gates (k)
This paper	3–9	1/2, 1/3, 1/4	200	162
Cavallaro and Vaya (2003)	3–9	1/2, 1/3	60.6	190
Batcha and Sha'ameri (2007)	5,7	1/2, 1/3	150	–
Campos and Cumplido (2006)	3–7	1/2, 1/3	70	175
Vennila et al. (2013)	3–7	1/2, 1/3	81	113
Niktash et al. (2006)	7, 9	1/2, 1/3	27, 10	–
Bissi et al. (2008)	4–9, 10–14	1/2, 1/3	0.337	1.314
Swaminathan et al. (2002)	7, 9	1/2, 1/3	3.125, 12.5	95
Benaissa and Zhu (2003)	7–10	1/2	101	172

Chang F, Onohara K, Mizuochi T, 2010. Forward error correction for 100 G transport networks. *IEEE Commun Mag*, 48(3):S48-S55.
https://doi.org/10.1109/MCOM.2010.5434378

Kim J, Yoshizawa S, Miyanaga Y, 2012. Variable wordlength soft-decision Viterbi decoder for power-efficient wireless LAN. *Integr VLSI J*, 45(2):132-140.
https://doi.org/10.1016/j.vlsi.2011.10.002

Moon TK, 2005. Error Correction Coding: Mathematical Methods and Algorithms. John Wiley & Sons, Inc., New Jersey, USA, p.487-490.
https://doi.org/10.1002/0471739219

Niktash A, Parizi HT, Bagherzadeh N, 2006. A multi-standard Viterbi decoder for mobile applications using a reconfigurable architecture. IEEE 64th Vehicular Technology Conf, p.1-5. https://doi.org/10.1109/VTCF.2006.176

Swaminathan S, Tessier R, Goeckel D, et al., 2002. A dynamically reconfigurable adaptive Viterbi decoder. Proc ACM/SIGDA 10th Int Symp on Field-Programmable Gate Arrays, p.227-236.
https://doi.org/10.1145/503048.503081

Vennila C, Patel AK, Lakshminarayanan G, et al., 2013. Dynamic partial reconfigurable Viterbi decoder for wireless standards. *Comput Electr Eng*, 39(2):164-174.
https://doi.org/10.1016/j.compeleceng.2012.12.009

Xiong L, Yao D, Tan Z, et al., 2004. Research on FPGA-based soft-decision Viterbi decoder for convolutional codes puncturation. *J Beijing Jiaotong Univ*, 28(5):36-39 (in Chinese).
https://doi.org/10.3969/j.issn.1673-0291.2004.05.009

Yoo W, Jung Y, Kim MY, et al., 2012. A pipelined 8-bit soft decision Viterbi decoder for IEEE802.11ac WLAN systems. *IEEE Trans Consum Electron*, 58(4):1162-1168.
https://doi.org/10.1109/TCE.2012.6414981