 By Syed Shahzad Shah, Saqib Yaqub, and Faisal Suleman, Chameleon Logics

SYSTEM DESIGNERS FIGHT NOISE WITH ERROR-CORRECTING CODES THAT ADD EXTRA INFORMATION TO A TRANSMITTED SIGNAL, INCREASING THE RECEIVER'S PROBABILITY OF RECOVERING CORRECT DATA.

Self-correcting codes conquer noise Part one: Viterbi codecs

FEC (FORWARD-ERROR-CORRECTION) techniques correct errors at the receiver end of digital communications systems. In contrast with error-detection and retransmission techniques, FEC requires only a one-way link, and its parity bits target both error detection and correction. **Figure 1** shows a typical FEC communication system. The transmitter encodes the information before modulating and sending the signal through the channel. The communications channel introduces noise such as AWGN (additive white Gaussian noise) to the transmitted signal. The receiver demodulates the corrupted information, which a decoder then processes, to retrieve the original information. You can realize the FEC techniques on block codes and convolutional codes. Some applications of FEC include CD and DVD players, high-definition TV, data-storage systems, wireless communications, satellite communications, and modem technologies.

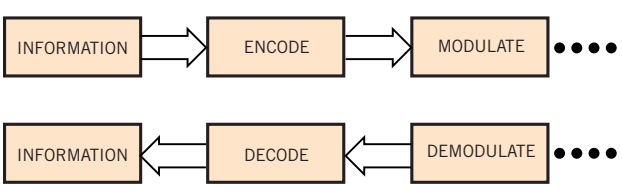
Designers have implemented FEC techniques in application-specific-standard products, ASICs, DSPs, and FPGAs. ("FPGA" here denotes a reconfigurable architecture.) Application-specific-standard products meet performance requirements but provide no flexibility. DSPs, on the other hand, provide flexibility but lack performance. Designers facing space and power constraints will go for an ASIC, but inflexibility and the demands for a faster time to market in some cases make this alternative unattractive. FECs suit FPGAs because they have efficient parallel architectures, you can reconfigure them without nonrecurring-engineering costs, their performance is always improving, and FEC cores for FPGAs are inexpensive.

You can implement FEC techniques for linear block codes (Reed-Solomon codecs) and convolutional codes (Viterbi codecs). To understand this general implementation, look at the hardware behind these error-correction processes. The **sidebar** "Viterbi-codec definitions" contains a Viterbi algorithm explained in terms of its hardware. You can also tailor algorithms for hardware efficiency. Part two of this article discusses the Reed-Solomon-codec implementation. Digital design with reconfigurable architectures is often a daunting task because good designs demand sound design practices as well as an intimate understanding of both the tools and the silicon. Also, reconfigurable architectures suit some applications, such as pipelined designs, and not others, such as pulsed-mode designs.

CONVOLUTIONAL ENCODERS

As previously discussed, an encoder at the transmitter end adds redundancy to the information being sent. **Figure 2** shows an implementation with a code rate of 1/2 and K=3. The figure shows that convolutionally encoding the data is fairly simple; you do it using shift registers and modulo-2 addition. The connections between the registers and the adders give rise to the characteristic of the code, and the change in connections therefore gives rise to a different code. Furthermore, a practical system must determine the validity of the data at the input and the output. The control unit serves this purpose. In

Figure 1



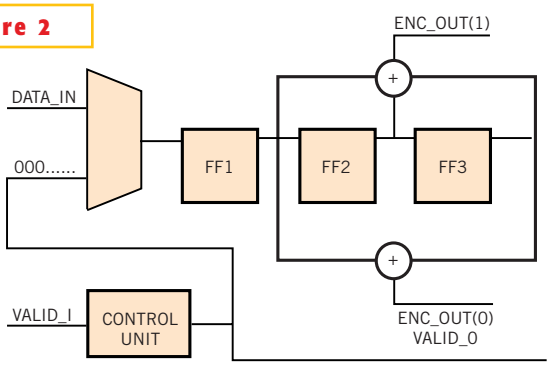
An error-correcting communication system adds information at the transmitter to fight noise-data corruption.

general, the encoder needs K−1 flush bits to ensure that the message shifts the full length. You can easily implement this type of encoder in any reconfigurable architecture. The polynomial generators are $g1=1+X+X^2$; $g2=1+X^2$.

HARD-DECISION VITERBI DECODING

The first step in the Viterbi-decoding algorithm requires calculation of the branch-metric (see **sidebar** "The Viterbi-decoding algorithm"). The branch metric is the distance from the received code word to all the possible branch words. You can derive the branch word from the trellis or state diagram; the branch word depends on the constraint length, the generator matrix, and the code rate. In this case, the possible branch words are 00,01,10, and 11. The distance from branch word 11 to the received sequence is called Ham_11, and others are similarly named. If you apply this logic to a truth table, you will realize that a half adder with simple logic implements the truth table in hardware. **Figure 3a** shows the hardware realization of this unit. FPGAs can realize the logic in **Figure 3b** by storing the hamming distances in ROM look-up tables. This technique minimizes the hardware required to implement a large number of branch words.

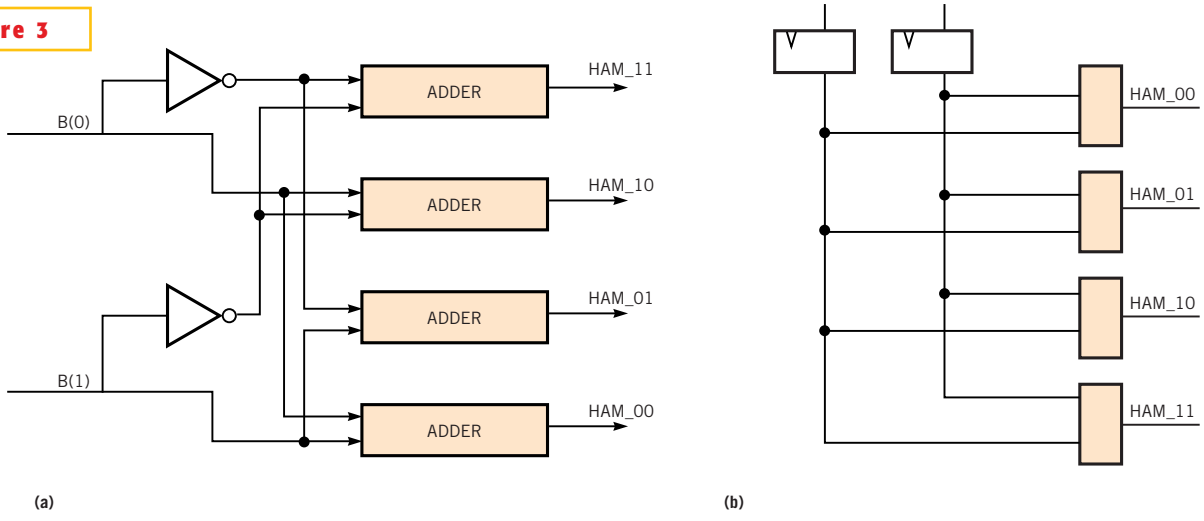
Figure 2



You can implement a Viterbi encoder with code rate 1/2 and K=3 with shift registers and modulo-2 addition.

The second step of the Viterbi decoding algorithm, the ACS (add-compare-select) unit, is the heart of the process

Figure 3



You can implement the Viterbi branch-metric-unit calculation with half adders (a) and ROM look-up tables (b) to store the hamming distances.

VITERBI-CODEC DEFINITIONS

The following terms are vital to the understanding of convolutional codes.

Constraint length: This number, called K, represents the number of k-bit shifts over which a single information bit can influence the encoder output. Simply put, constraint length is the number of bits that the encoder uses to encode n bit. The computational requirements grow exponentially as a function of constraint length, so, in practice, the constraint length is limited to 9 bits or less. A greater value of K means more redundancy, which therefore results in better noise immunity. Code rate is fixed, and K varies to control the redundancy.

Code rate: K bits enter the Viterbi encoder; and n bits leave the encoder; therefore, the code rate is K/n. For example, in rate 1/2 code (1 bit in, 2 bits out), each bit entering the encoder leaves the encoder with 2 bits. Typical values are: 1/2, 1/3, and 2/3. Error-correcting capability and hardware complexity increase as K/n decreases.

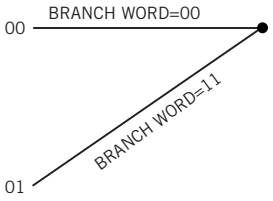
Hard-decision/soft-decision decoding: The demodulator gives information to the decoder. Hard-decision decoding means that the demodulator is quantized to two levels: zero and one. If you derive more than two quantization levels from the demodulator, then the decoder is soft-decision decoding. Hard-decision decoding infers a performance loss of 2 to 3 dB over soft-decision decoding.

Generator polynomial: A generator polynomial specifies the encoder connections. Each polynomial is of degree K−1 or less and specifies the connections of the shift register and the modulo-2 adder.

and dictates the performance of the decoder. The ACS operation for each new state in the trellis performs the addition, comparison, and selection of the smallest path metric. Digital realizations represent path metrics as fixed-point values, and, therefore, path-metric normalization is necessary to prevent overflow for hard decisions and underflow for soft decisions during the updating and accumulation of the path metrics. Different methods of normalization exist (**Reference 1**). Fixed-shift methods exploit an important property of path metrics in an ACS unit. That is, the magnitude of the path metrics is unlimited, yet a fixed number limits the difference between them. The following equations can help you calculate this fixed number: Δ=code rate*(K−1), and BOP=ceil($\log_2(\Delta+1)$) +1, where BOP is the bit width of the path metric, code rate is the inverse of K/n, and ceil is the function. For code rate=1/2, K=3, and BOP=3, these values mean that the maximum theoretical difference of path metrics does not exceed 7. Also, an extra bit at the most significant bits is necessary in this case to maintain precision. When all the path metrics have 1 at the most significant bits, you can use simple logic to normalize the ones to zeros. This normalization has no effect on these results, because you are interested only in the differences between these metrics. For example, for the arbitrary path metrics 16, 19, 20, and 16, the results are 0, 3, 4, and 0.

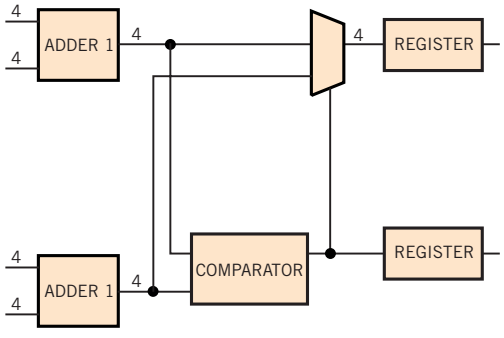
A portion of the trellis diagram shows how you can extract the resulting hardware (**Figure 4**). The trellis shows that two paths reach each node or state. You add the hamming distance from the BMU to the path metric. Therefore, each node needs two 4-bit adders to directly translate the trellis node into hardware. **Figure 5** shows that the upper inputs to the adder are the hamming distances from the BMU with zeros added. The lower bits to the adder are the path metrics (previous branch metrics). A comparator then compares the resulting path metrics, and the lesser one is the output from the ACS unit. To exactly duplicate the trellis diagram and provide feedback, you must place registers at the outputs.

Figure 4



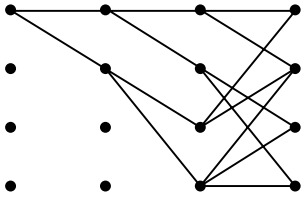
A portion of the trellis diagram shows that two paths reach each node or state.

Figure 5



Translating the trellis node into hardware requires two 4-bit adders and a comparator.

Figure 6



The trellis diagram for a constraint length K=3 has four states.

The constraint length of K=3 has four states, and for each state you need four slices of hardware. You can obtain the connection between these slices by inspecting the trellis shown in **Figure 6**. Your hardware implements the nodes that two paths enter. For example, node 1 calculates the path metric by feeding back its own value and feedback from node 3. The connections in **Figure 6** result in the hardware shown in **Figure 7**. These connections ensure that you correctly add the previous path metrics to the current branch metric. The outputs from the ACS slices go to another unit, which simply finds the state having the smallest path metric. That state is the output of the ACS unit.

MODULO-NORMALIZATION TECHNIQUE

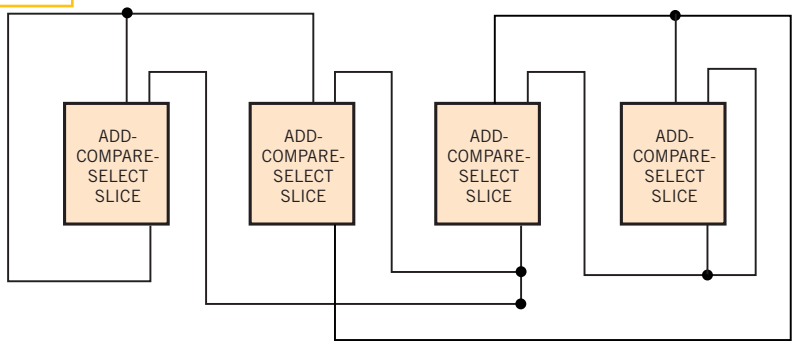
You must represent numbers in a 2's-complement format to apply the modulo- normalization technique. The metric is normalized inside the 2's-complement adders in the ACS slice. If M_i represents the surviving path metric, then you can represent the normalized path metric, $\overline{M}_i$ (output of the adder), as $(M_i+C/2)\text{Mod } C-C/2$. The number of bits that the adder requires in this case remains the same. This situation is possible because you can normalize the hamming distances. This method suits both hard- and soft-decision encoding. However, for soft decisions, you need not normalize the Euclidean distance. In **Figure 5**, you cannot use the same comparator for this calculation, either signed or unsigned; however, a 2's-complement subtracter works. The advantage of this method is that it eliminates the logic that normalization requires in a fixed-shift method; therefore, it improves both FPGA speed and the amount of chip area consumed.

The cost of an FPGA relates directly to its resources; therefore, it is important to note that the coding style, synthesis tool, back-end-implementation software, and FPGA architecture directly affect the pricing. For ACS units with larger values for K, you need 2^{K-1} ACS slices. If performance is not an issue, you can use two ACS slices for K=9 and code rate=$^1/_2$; however, you'll need more complex control.

FPGAs efficiently implement adders. Higher code rates and larger values of K need 5- or 6-bit or even larger adders for the ACS slices. Adder implementation depends on the FPGA architecture. To accelerate and condense arithmetic functions, such as adding and counting, Xilinx (www.xilinx.com) and Altera (www.altera.com) FPGAs contain dedicated, hard-wired, fast carry chains. The

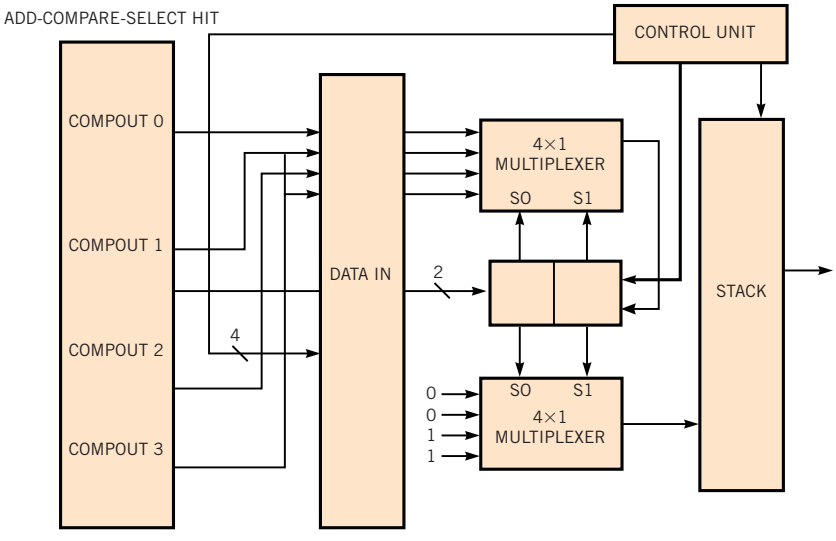
advantages of these carry chains are that they are faster than most other fast-carry schemes and using them eliminates the need to use the FPGA's look-up table. Hence, these units optimize both speed and size. Implementing ripple-carry adders with dedicated carry units is the best choice for adders with as many as 32 bits. You can use other fast-carry schemes together with the segments of ripple-carry adders for adders with more than 32-bits. Atmel (www.atmel.com) and Actel (www.actel.com) ripple-carry adders are the simplest and most compact, but a carry that must ripple through the design of the adder seriously limits their performance. A carry-select adder, on the other hand, improves speed by 40 to 90% by performing additions in parallel and reducing the maximum carry path.

Figure 7



Connections for multiple add-compare-select slices ensure the correct addition of the previous path metrics with the current branch metric.

Figure 8



The 16×4 RAM stores decisions from the add-compare-select unit for each unit of time, and the control unit generates the addresses and performs other functions, such as read/write and stack operations.

PATH MEMORY AND DECISION TRACE-BACK

The ACS unit at each unit of time gives you information about the survived path of the two paths entering each node. A 1 at this output means that you must retain the lower path of **Figure 4.** The ACS unit repeats until the end of the trellis (the third step of the Viterbi Algorithm), and you keep saving the information that the ACS provides.

Two methods exist for saving the information and making a decision based on that saved information. The first of these methods is the register exchange, which implements the trellis structure directly in hardware. It requires a bank of $L*2^{K-1}$(depth of trellis) registers. Each bank is associated with a state. You interconnect these registers in the same manner as you do the ACS elements. You determine the hardware requirement by multiplying L by the number of states. Therefore, you need 60 multiplexed registers for K=3 and code rate=1/2 if L=15. An FPGA with two registers per logic element consumes 30 logic elements. This method requires more FPGA resources as the trellis depth, L, increases, so it is unsuitable for larger values of K that require more trellis depth.

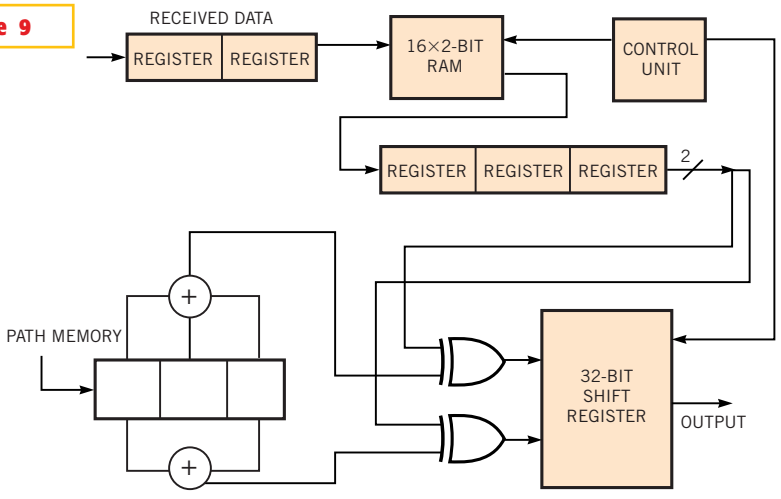
The second method is the trace-back-based decision unit, which is more efficient and requires fewer FPGA resources. You can implement the storage in RAM. The most efficient way to implement memory in an SRAM FPGA is by using look-up tables. You can use them as RAMs or ROMs or for implementing combinational-logic functions. In a Lucent (www.lucent.com) FPGA, for example, each logic element can implement two RAM or ROM arrays: a single 16×4 element or two 16×2 memory blocks. You can use multiple logic elements to implement other array sizes, such as 16×8, 32×4, and 64×8. Xilinx FPGAs implement 16×2 RAM or 32×1 in one configurable-logic block with support for larger memories. The EDA tools make it easy to implement RAMs in FPGAs.

The 16×4 RAM stores decisions from the four-ACS unit, or comparator output, at each unit of time (**Figure 4**). This RAM is now called the path memory. Thus, path memory is the set of sequences of decision bits that lead to current state at each unit of time. The size of the path memory is a direct replica of the trellis and, in this case, is limited to five constraint lengths. Also, higher code rates generally require greater path-memory depth than low rates. The row forms the time, and the state forms the column. This method of implementation is more efficient than moving an entire path from one memory location to another. The control unit in **Figure 8** generates the addresses and other functions,

such as read/write and stack operations.

Once you store all the decision bits, the hardware is ready to give the original information. The trace-back in the trellis is equivalent to tracing back in the path memory. It is best to start the trace-back with the lowest state metric, which the 2-bit register stores at that particular time. During the subsequent trace-back operation, the 2-bit register updates by appending to it the selected decision bit from the RAM. The 2-bit register forms the selection lines of the 4×1 multiplexer, which outputs the original information bit at each time. The LIFO (last-in, first-out) stack stores this original information, and the stack reorders and delivers the original information at the decoder output. You can easily implement the LIFO stack in a 16×1 RAM. In an SRAM-based FPGA, this implementation results in a single look-up table.

Figure 9



The optional bit-error-rate unit determines the beginning of the branch word in the received sequence.

BIT-ERROR-RATE UNIT

"BER" (bit-error rate) refers to the number of errors in a data transmission divided by the total number of bits sent. For example, 10 kbits are encoded, and 20 kbits enter the noisy channel. Say that the noisy channel incurs 2000 errors, and you feed the corrupted data to the Viterbi decoder. If the decoder corrects 1700 errors, the BER=0.03. Ideally, you want the BER to be 0. The BER unit is optional but useful, and it determines the beginning of the branch word in the received sequence. This information helps you determine the start of the valid data at the receiver end. **Figure 9** shows an example of a BER unit that you can implement.

When the quantization level of the demodulator output is greater than two, the decoding is called soft-decision decoding. The quantization levels are usually limited to eight (n=3 bits), because more than eight levels provide little extra gain but greatly increases the complexity of the decoder. The output from the demodulator can be either in a binary-offset format (the unsigned binary numbers) or a 2's-complement format or in sign-magnitude format. For n=3, the 2's-complement format gives the output a range of −4 to +3. For hard-decision decoding the branch metric is the hamming distance; for soft-decision decoding, it can be the linear or Euclidean distance. The Euclidean distance for code rate, 1/C, is:

THE VITERBI-DECODING ALGORITHM

Qualcomm co-founder Andrew Viterbi discovered and analyzed the Viterbi-decoding algorithm. He derived the algorithm from the maximum-likelihood concept of decision theory. Generally speaking, the decoder chooses a specific sequence as a transmitted word if the likelihood of the word's duplicating the original transmission is greater than the likelihood of all the other possible transmitted words' matching. You break down the algorithm in the computational efforts defined below to reach the possible transmitted words.

Branch word: A branch word is the output from the encoder that results from the transition of one encoder state to another.

Branch metric: A branch metric is the distance between the received sequence and the branch word. This distance is either hamming (for hard-decision decoding) or Euclidean (for soft-decision decoding).

Path metric: An accumulation of branch metrics forms the path metric.

Trellis: The trellis diagram is the most important concept in understanding the Viterbi algorithm. The trellis shows the states, or so-called nodes, of the convolutional encoder at different times.

The Viterbi algorithm comprises the following steps:

- Find the path metrics of every path at each node by adding the appropriate branch metric to its corresponding survived path metric.
- Compare the paths entering each node (two, in the case of K=3) and select the one with the smallest path metric. This path is the surviving path metric. Perform this step in parallel for 2^{K-1} states.
- Repeat this procedure until you reach the end of the trellis.

As you reach the end of the trellis, all the paths merge to a common state. Hence, only one path survives; tracking that path, you can obtain the original information. Usually, the depth of the trellis is five times the constraint length. An increase in computational resources with no significant performance advantage deepens the trellis.

REFERENCE

Viterbi, Andrew, "Error bounds for convolutional codes and an asymptotically optimum decoding Algorithm," *IEEE Transactions on Information Theory*, Volume IT 13, April 1967, pg 260.

$$LOCAL\_DISTANCE = \sum_{n=0}^{C-1} [SD_n - G_n]^2,$$

which, simplifies to

$$LOCAL\_DISTANCE = \sum_{n=0}^{C-1} SD_n G_n.$$

where, SD_n=soft-decision input, and G_n=the branch word for each path.

For a rate of 1/2, the above equation becomes:

$$LOCAL\_DISTANCE = SD_0 G_0 + SD_1 G_1.$$

Bits are transmitted as signed antipodal signals. That is, zero is transmitted as a positive voltage, and one is transmitted as a negative voltage. So you can simply reduce the above equation to the sum and difference of the received inputs.

For R=1/2, you need to calculate only two branch metrics, because the other two are simply the inverse of that calculation. So, in each slice of the ACS, you use one adder and one subtracter. In simplifying the above equation you ignore a minus sign, so you must maximize the path metric; that is, you must select the larger of the two branch metrics. The rest of the hardware details are the same as those for the hard-decision decoder.

FPGAs from Xilinx, Altera, Actel, and Atmel are general-purpose designs, meaning that you can implement any feasible digital system with these devices. Feasible digital systems include Viterbi decoders, which are important components of communication systems. Another reconfigurable device for Viterbi-decoder implementation is Chameleon Systems' (www.chameleonsystems.com) CS2000 family of reconfigurable communication processors. These devices include a 32-bit RISC processor, blocks of embedded memory, a reconfigurable-processing fabric, and many I/O pins. Because FEC relates to communication systems, chameleon processors are well-suited for FEC algorithms' hardware implementation. You can also implement a soft-decision Viterbi decoder in the CS2000's reconfigurable-processing fabric (**Figure 10**).

Viterbi techniques are just one of the error-correction schemes that advanced communications systems use. Part two of this article investigates the theory and implementation of Reed-Solomon codecs.□

References

1. Shung, Bernard C, Paul H Siegel, Gottfried Ungerboeck, and Hemant K Thapar, "VLSI Architectures for Metric Normalization in the Viterbi Algorithm," IBM Corp, IEEE-1990.

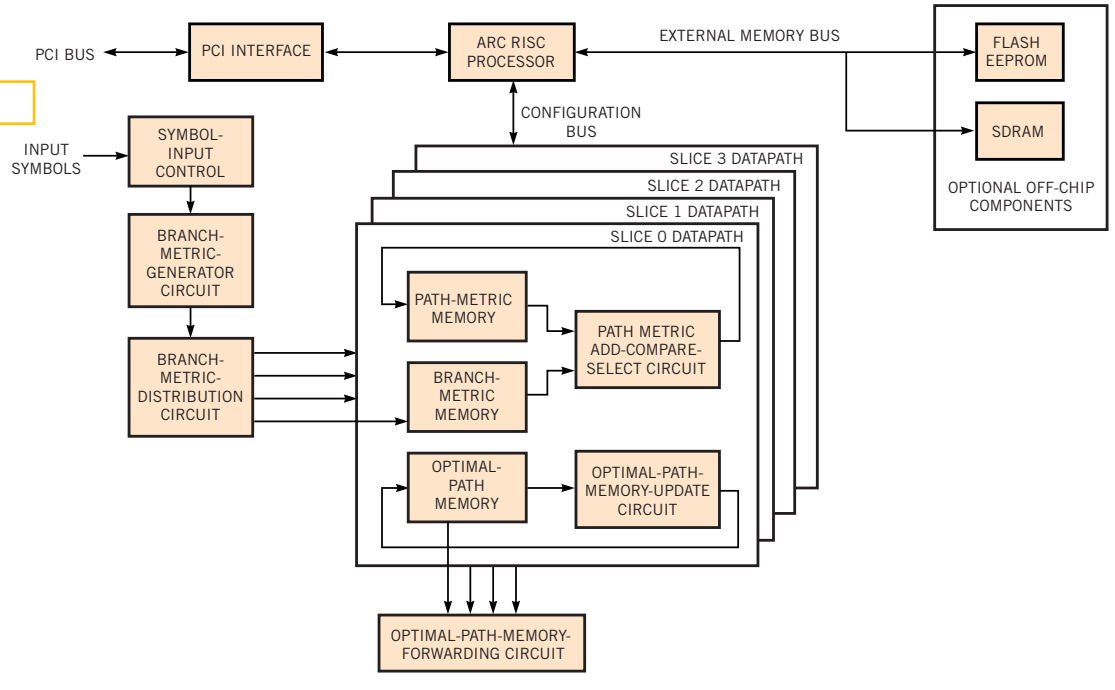
Authors' biographies

Syed Shahzad Shah is the CEO of Chameleon Logics (Islamabad, Pakistan). He has a master's degree in electrical engineering and more than six years of experience in VLSI design. His areas of interest and expertise are forward-error correction, ASICs/FPGAs, DSP algorithms in FPGAs, and digital-communication systems.

Saqib Yaqub holds a bachelor's degree in electrical engineering and works for Chameleon Logics as an FPGA core engineer. He has more than one year of experience in designing FPGAs and has experience in forward-error correction and DSP algorithms in FPGAs.

Faisal Suleman holds a bachelor's degree in electrical engineering and works for Chameleon Logics as an FPGA core engineer. He has more than one year of experience in designing for FPGAs and has experience in forward-error correction and DSP algorithms in FPGAs.

Figure 10



You can use Chameleon's CS2000 reconfigurable-processing fabric to implement a soft-decision Viterbi decoder.