Implementing Logic with the Embedded Array

in FLEX 10K Devices

October 2000, ver. 2 Product Information Bulletin 21

Introduction

Altera's FLEX® 10K devices are the first programmable logic devices (PLDs) to contain embedded arrays, which allow designers to quickly create, prototype, and debug complex designs. Unlike embedded functions in a gate array, the FLEX 10K embedded array is fully programmable, giving the designer complete control over the functions programmed in the embedded array. The FLEX 10K embedded array is composed of a series of embedded array blocks (EABs), which can be used to implement memory and logic functions.

This product information bulletin describes the capabilities of the FLEX 10K embedded array, and how designers can use the EAB to implement logic in a variety of applications. The following topics are discussed:

- Logic cells vs. EABs
- Configuring the EAB as a look-up table (LUT)
- Embedded vs. distributed RAM
- Applications

Logic Cells vs. EABs

Logic cells, which contain combinatorial logic and registers, can implement relatively simple functions such as one bit of an adder or a small multiplexer. To implement complex, high fan-in functions, the function must be divided among multiple logic cells, which are connected using additional logic. The number of logic cells required increases rapidly as the function becomes more complex.

In contrast, the FLEX 10K embedded array implements complex functions in a single logic level, resulting in more efficient device utilization and higher performance. Thus, many complex functions implemented in an EAB will occupy less area on a device, have a shorter delay, and operate faster than functions implemented in logic cells.

An EAB can implement any combinatorial function, such as a 4 × 4 multiplier, provided the function does not exceed the permitted number of inputs and outputs to the EAB. Depending on its configuration, an EAB can have 8 to 11 inputs and 1 to 8 outputs, all of which can be registered for pipelined designs. See Table 1.

Table 1. Inputs and Outputs per EAB

Inputs	Outputs
8	8
9	4
10	2
11	1

EABs can be cascaded to implement functions that require more inputs or outputs than are available in a single EAB. Each EAB can have a maximum of 11 inputs and 1 output. Therefore, a function with 11 inputs and 2 outputs is divided into two EABs, so that each EAB has 11 inputs and 1 output.

Reconfiguring the EAB for a different number of inputs and outputs does not affect its performance. The delay in an EAB remains constant, provided the function fits into the EAB (i.e., has a permissible number of inputs and outputs). Likewise, the delay in each EAB is the same for two functions that each fit into an EAB. For instance, the delay in the EAB for a 6-input function and the delay in the EAB of an 8-input function are the same.

In addition, the timing performance in an EAB does not change as its configuration size changes. EABs can be cascaded to form RAM blocks up to 2,048 words without affecting performance. The EAB RAM size is flexible and can be configured as any of the following sizes: 256 × 8, 512 × 4, 1,024 × 2, or 2,048 × 1. The appropriate configuration size depends on the function to be implemented; for instance, an EAB is configured as 256 × 8 to implement an 8-input, 8-output function. Larger RAMs are created by combining multiple EABs. Thus, two 256 × 8 RAMs can be combined to form a 256 × 16 RAM without a timing penalty.

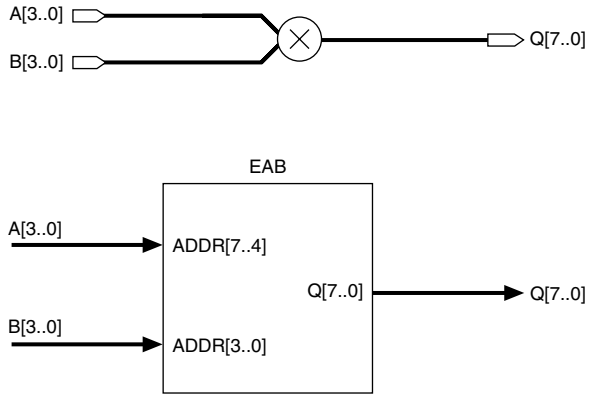
Configuring the EAB as a Look-Up Table

Logic functions are implemented by programming the EAB during configuration with a read-only pattern, creating a large LUT. The pattern can be reconfigured during device operation to change the logic function. The LUT looks up the results of the functions rather than using algorithms to calculate them.

When a logic function is implemented in an EAB, the input data is driven in on the address input of the EAB. The result is looked up in the LUT and driven out on the output port. Using the LUT to find the result of a function is faster than using algorithms implemented in general logic.

For example, in a 4 × 4 multiplier with two 4-bit inputs and one 8-bit output implemented in an EAB, the two input buses drive the address inputs of the EAB. The data output of the EAB drives out the product. See Figure 1.

Figure 1. Implementing a 4 × 4 Multiplier in an EAB



The EAB acts as a LUT to find the product. Table 2 shows part of the pattern used to implement a 4 × 4 multiplier. Values are shown in hexadecimal radix.

Table 2. Portion of EAB Pattern for Implementing a 4 × 4 Multiplier

ADDR[7..4] (Input A)	ADDR[3..0] (Input B)	Q[7..0] (Product)
0	0	00
0	1	00
2	4	08
7	2	0E
A	A	64
A	B	6E

Embedded vs. Distributed RAM

FLEX 10K embedded RAM implements logic functions more efficiently than distributed RAM. Distributed RAM, as used in field-programmable gate arrays (FPGAs), allows the designer to use a particular array of memory cells either as part of the general logic array or as addressable RAM. However, using distributed RAM provides only small RAM blocks such as 16 × 2 or 32 × 1. Using distributed RAM for applications larger than 32 × 1 results in lower performance and lower device utilization. To create larger RAM blocks, the small RAM blocks must be interconnected using additional logic cells. However, adding logic cells can cause less predictable delays, routing problems, and can reduce the amount of available logic for implementing other functions. Therefore, there is no advantage gained from implementing logic functions with distributed RAM than with logic cells.

In contrast, FLEX 10K devices dedicate a portion of the device to embedded RAM. Embedded RAM is implemented in the EAB, which is a large block of flexible RAM. Altera's MAX+PLUS II development software automatically cascades EABs to implement blocks of RAM larger than 2,048 × 1. Because the EAB is inherently a large RAM block, the EAB can implement complex logic functions in a single logic level, so additional logic cells are not required. FLEX 10K devices can offer as much as 24 Kbits of RAM without sacrificing logic capacity. Therefore, implementing logic functions with embedded RAM in FLEX 10K EABs results in higher resource utilization and predictable performance.

Manufacturers of distributed-RAM FPGAs claim that embedding large blocks of RAM into a programmable device is inefficient because die area is wasted if a design does not use RAM. However, EABs that are not used as memory will be used as logic, and most designs will contain some complex logic functions that can be implemented by EABs.

Applications

EABs can be used for a variety of specialized logic applications, including:

- Symmetric multiplier
- Asymmetric multiplier
- Constant multiplier/vector scalar
- Digital filter
- Two-dimensional convolver
- State machine
- Transcendental functions
- Waveform generator
- 8-bit-to-10-bit encoder

Symmetric Multiplier

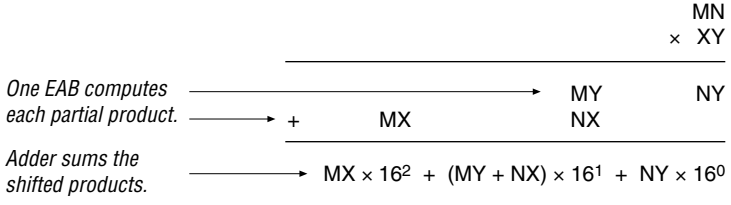
A symmetric multiplier multiplies two inputs of the same width. An EAB can easily implement a 4 × 4 multiplier, which has two 4-bit inputs and one 8-bit output. The EAB drives the two multiplicands into the address input and reads the product from the data output. For example, to multiply the number 2 by the number 4, 4 bits of the address input represent the number 2, and the other 4 bits represent the number 4. Because multiplication is commutative, address locations 24 and 42 both store the value 08.

Designers can create larger multipliers by using parallel multipliers or time-domain-multiplexed multipliers to combine EABs.

Parallel Multiplier

A parallel multiplier uses multiple EABs to generate all partial products in parallel. A parallel multiplier uses 4 EABs for an 8 × 8 multiplier (i.e., 1 EAB for each partial product). Each of the 4 EABs simultaneously processes a portion of the input to generate a 4 × 4 product, yielding a total of four 4 × 4 products. A two-stage adder implemented in the logic cells produces the final result. For example, MN is multiplied by XY in Figure 2. Each letter in the multiplicands represents four bits of the input; M represents the four most significant bits (MSBs), and N represents the four least significant bits (LSBs). Before summing the products, the products are multiplied by 16^n (where n = 0, 1, 2...) to account for their relative significance in hexadecimal radix. Larger multipliers are created with additional EABs.

Figure 2. 8 × 8 Multiplier Implemented in an EAB



Time-Domain-Multiplexed Multiplier

A time-domain-multiplexed multiplier uses a single EAB to generate all the partial products on different Clock cycles. Multiplexers at the input of the EAB route the appropriate inputs into the EAB, and the EAB calculates each partial product at a different time. After each multiplication is performed, the products are multiplied by 16^n (i.e., shifted left) to account for their relative significance in hexadecimal radix. An accumulator adds the four partial products to produce the final result. For an 8 × 8 multiplier, the time-domain-multiplexed multiplier requires four Clock cycles. The required number of Clock cycles can be reduced by using more EABs. Larger multipliers are created with additional EABs, or by increasing the required number of Clock cycles.

Asymmetric Multiplier

An asymmetric multiplier multiplies two inputs of different widths. For example, one EAB can implement a multiplier that multiplies a 2-bit input by a 6-bit input to create an 8-bit output. Like symmetric multipliers, larger asymmetric multipliers are created using parallel multipliers or time-domain-multiplexed multipliers to combine multiple EABs. Each EAB computes one of the partial products, and adders are used to sum the products. Therefore, a 10 × 6 multiplier can be created from 5 EABs. Figure 3 shows how each EAB in an asymmetric multiplier computes a partial product.

Figure 3. Asymmetric Multiplier Implemented in an EAB

Values are shown in hexadecimal radix.

						×	LMN XY
			LY		MY		NY
+		LX		MX		NX	
$LX \times 16^3$	+	$(LY + MX) \times 16^2$	+	$(MY + NX) \times 16^1$	+	$NY \times 16^0$	

See *Application Note 53 (Implementing Multipliers in FLEX 10K Devices)* for more information about implementing multipliers.

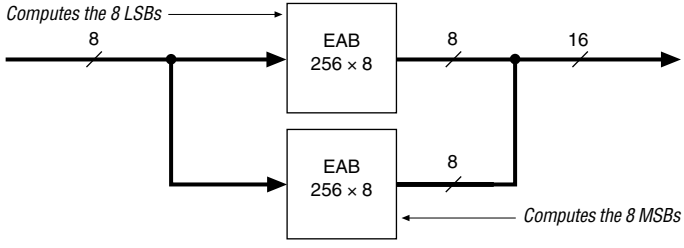
Constant Multiplier / Vector Scalar

The embedded array can efficiently implement constant multipliers. The constant multiplier is used for datapath applications such as video and digital signal processing (DSP) that require a series of numbers (a vector) to be multiplied or scaled by a constant. The value of the constant determines the EAB pattern used to implement the function.

Depending on the width and the required precision of the data, one or more EABs can be used to perform the multiplication. For instance, one 256 × 8 EAB can multiply a 4-bit number by 13 (a 4-bit value) without any truncation. The 4-bit input drives the address input, and the output appears on the data output.

The required precision of the output must be determined before multiplying larger numbers in an application. If the output does not require full precision, the output can be truncated to minimize the number of EABs needed to calculate the result. If precise output is required, multiple EABs must be used. For example, if a series of 8-bit variables are multiplied by an 8-bit constant, the result could be as large as 16 bits. If only 8-bit precision is required, one EAB can calculate the product because the 256 × 8 EAB has 8-bit-wide input and output ports. If full precision is required, one EAB calculates the 8 MSBs, and another EAB calculates the 8 LSBs. Figure 4 shows how a constant multiplier is implemented in multiple EABs.

Figure 4. Constant Multiplier Implemented in Multiple EABs

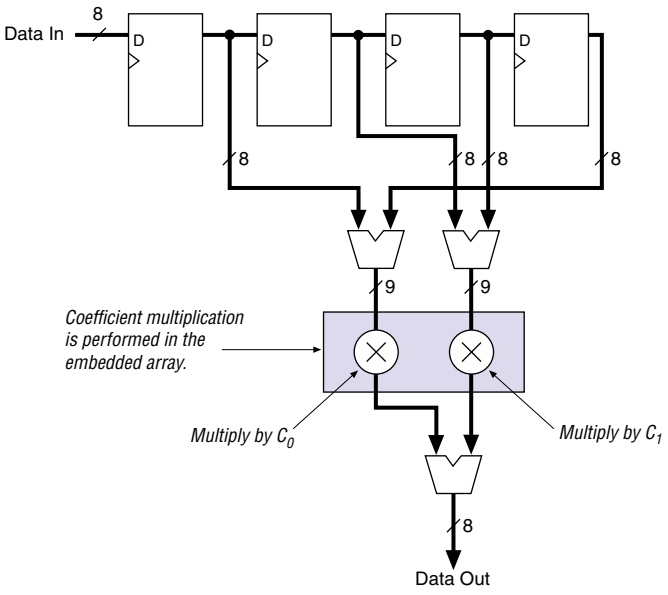


Digital Filter

Digital systems are being used more frequently for filtering applications. A common digital filter is the finite impulse response (FIR) filter, which shifts incoming data through a series of registers. The output of each bank of registers is called a tap. The output per time period is the sum of all taps, which is calculated by multiplying each tap by a coefficient and summing the products.

The filter's frequency response is determined by the value of the coefficients used in the design. In a linear phase response FIR filter, the coefficients are symmetric, i.e., the coefficient for tap n is equal to the coefficient for tap $(m - n - 1)$, where m is the total number of taps. For example, if there are 8 taps, the coefficients for tap 1 and tap 6 are equal. Because the coefficients for tap 1 and tap 6 are equal, only half the number of multipliers are needed to calculate the output per time period; using the distributive property of multiplication, the taps with the same coefficients are summed before multiplication, e.g., $ac_0 + bc_0 = c_0(a + b)$, where c_0 is a coefficient. Figure 5 shows a schematic diagram of a 4-tap FIR filter.

Figure 5. 4-Tap FIR Filter



The EAB, configured as a LUT, can implement a FIR filter by performing the coefficient multiplication for all taps. The multiplication for all taps is spread across several EABs, with each EAB calculating the partial products for 1 bit of each tap. For example, EAB 0 calculates the partial products for bit 0 of each tap. Then, the EAB outputs are summed by an adder in the logic array. The FLEX 10K carry chain is designed to implement fast, compact adders.

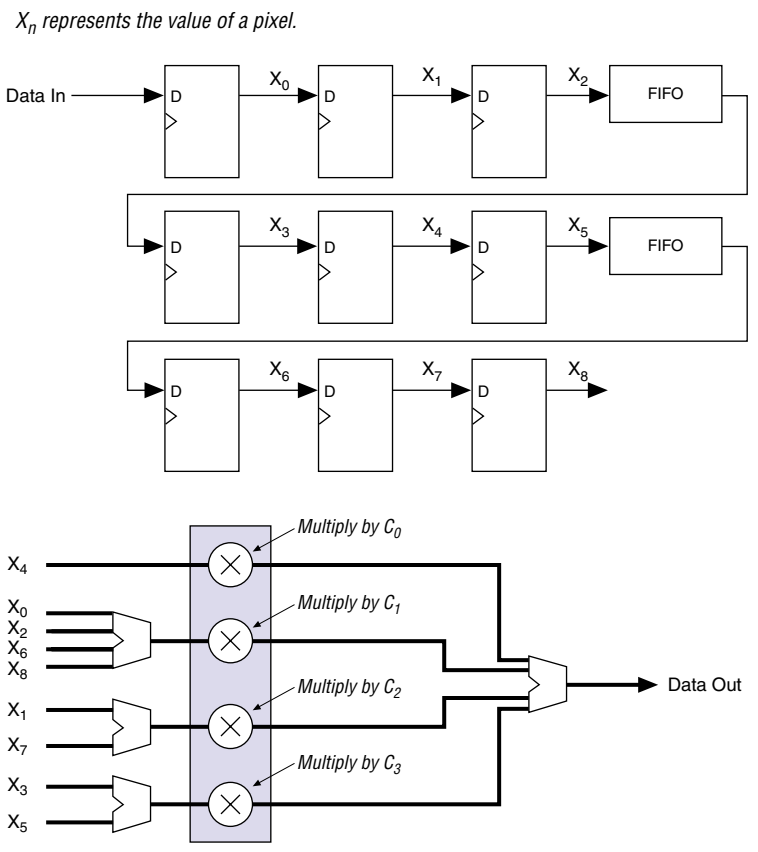
The required precision on the output and the number of taps in the FIR filter determine the EAB configuration used to implement the FIR filter. For 8-bit precision on the output, each EAB is configured with 8 outputs. The number of taps in the FIR filter determines the number of inputs required for each EAB; if the coefficients are symmetric, only half the number of inputs are required because the filter can sum the taps with the same coefficients before multiplying. Thus, using EABs with 8 inputs implements a FIR filter with a maximum of 16 taps.

Implementing a FIR filter with an embedded array can be more efficient than implementing a FIR filter with logic elements (LEs). An EAB has up to 8 inputs and 8 outputs, and could implement a 16-tap FIR filter without using complex logic to compute the coefficient multiplication. An LE has only 4 inputs, and would require multiple levels of logic to implement a FIR filter that required more than 8 taps. The EAB can be reconfigured on-the-fly, allowing the coefficients used in the FIR filter to be changed without disturbing the operation of the rest of the device.

Two-Dimensional Convolver

The embedded array can efficiently implement two-dimensional convolvers, which are used to process video images. For example, the convolver sharpens the edges of a picture for output in a technique called edge enhancement. The convolver processes the video information in small pieces, such as a 3 × 3 matrix, and then multiplies each pixel in the matrix by a constant coefficient. Because the coefficient values are usually symmetric, the number of multipliers needed is reduced by summing the multiplicands with the same coefficient before multiplying. The new value of the center pixel is the sum of all the matrix multiplications. Figure 6 shows a block diagram of a two-dimensional convolver.

Figure 6. Two-Dimensional Convolver



In general, the convolver and the FIR filter process data in a similar manner. FIR filters process a one-dimensional stream of data, and do not require first-in-first-out (FIFO) buffers for storing the data. Convolvers process a two-dimensional matrix of data, and the FIFO buffers store the data that is driven in from the inputs. The FIFO buffers are implemented with EABs. In the convolver implementation shown in Figure 6, two line FIFOs buffer each line as it is driven in from the external source. The depth of the FIFO buffer equals the width of the video matrix.

Like a FIR filter, the convolver is implemented with an EAB configured as a LUT that performs the coefficient multiplication. Four taps are required in Figure 6. Because the number of inputs to the LUT equals the number of taps required, only 4-input LUTs are required to implement the convolver. This convolver can be implemented in a FLEX 10K LE, which has four inputs.

Depending on the type of video processing desired, some of the tap coefficients may be equal. In Figure 6, the coefficient of the 4 taps (X_0, X_2, X_6, X_8) is the same (C_1); therefore, the outputs of the 4 taps are summed before multiplication. If 8-bit data is convolved, the sum is 10 bits. For 10-bit precision on the input, 10 LUTs are required. Each of the 10 LUTs requires eight outputs for 8-bit precision on the output.

State Machine

The embedded array can also be used to implement highly complex state machines. As a state machine becomes more complex (i.e., has additional transitions), the number of LEs required to implement the state machine increases, but the number of EABs required remains constant. The number of EABs required to implement a state machine is simply a function of the number of states, inputs, and outputs to the state machine. Therefore, the same number of EABs is required for two state machines with a different number of transitions but with the same number of states, inputs, and outputs.

The embedded array can implement general-purpose and limited-transition state machines. General-purpose state machines can have complex transitions between states, but in turn have only a finite number of states. Limited-transition state machines can implement more states in a given amount of logic, but consequently cannot have very complex transitions.

General-Purpose State Machine

The embedded array effectively implements general-purpose state machines with very complex transitions between states. The number of EABs required to implement the state machine does not change if the transitions become more complex.

The address input to the EAB is a combination of bits representing the inputs to the state machine and the current state. For example, in a 16-state, 4-input, 4-output state machine, signals representing the 4 inputs to the state machine drive `ADDR[7..4]`, and signals representing the current state drive `ADDR[3..0]`. Each address input to the EAB contains two fields: the outputs for the current state and state bits that indicate the next state. To design a Moore state machine, the design uses the input registers of the EAB. To design a Mealy state machine, the design uses LEs to register only the address bits that represent the current state. Figure 7 shows the implementation of a 16-state, 4-input, 4-output Moore state machine.

Figure 7. Moore State Machine Implemented in an EAB

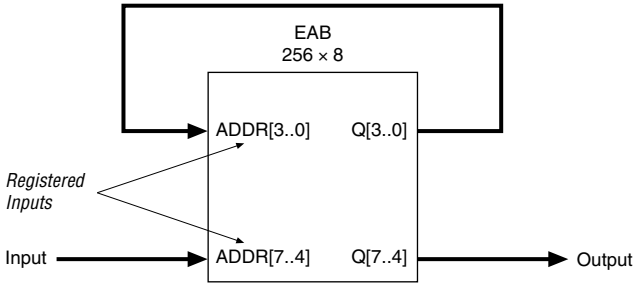
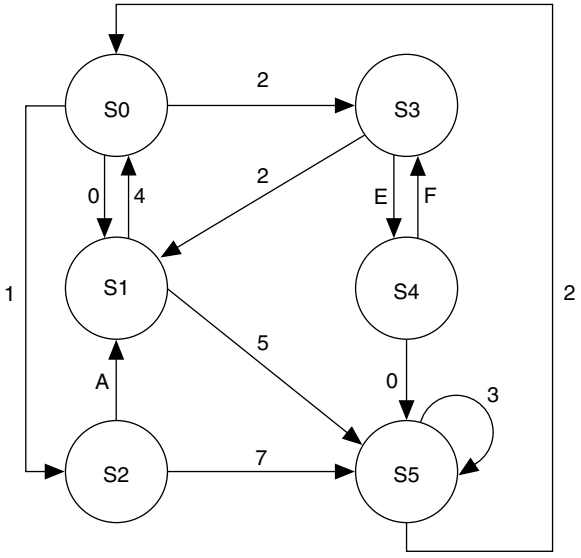


Figure 8 shows a state machine implemented in a portion of an EAB. The contents of the table control the behavior of the state machine. For example, in state `0` with state machine inputs equal to `0`, the state machine transitions to state `1`; in state `1` with state machine inputs equal to `5`, the state machine transitions to state `5`. These transitions are indicated in the first and fifth rows of the table, respectively.

Figure 8. State Machine Inputs Implemented in a Portion of an EAB

State machine input and output values are shown in hexadecimal radix.



State	State Machine Inputs ADDR[7..4]	Current State ADDR[3..0]	Outputs Q[7..4]	Next State Q[3..0]
S0	0	0	0	1
S0	1	0	0	2
S0	2	0	1	3
S1	4	1	3	0
S1	5	1	4	5
S2	A	2	1	1
S2	7	2	A	5
S3	2	3	C	1
S3	E	3	7	4
S4	0	4	2	5
S4	F	4	F	3
S5	3	5	2	5
S5	2	5	4	0

The size of the state machine's required memory is calculated from its memory width and memory depth. Memory width is a function of the number of outputs and the number of states; memory depth is a function of the number of inputs and the number of states.

Memory width = $Q + C (\log_2 (S))$

Memory depth = $2^{(D + C (\log_2 (S)))}$

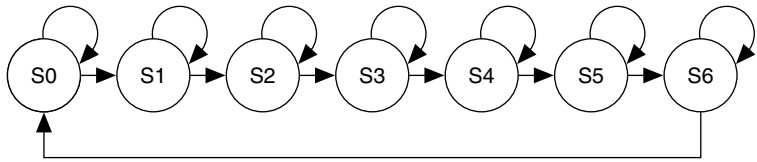
where Q = Number of outputs
C = Ceiling (The ceiling function returns the next highest integer value, i.e., ceiling (1.0) = 1, ceiling (1.1 ... 1.9) = 2.)
S = Number of states
D = Number of inputs

If the required memory space is larger than can fit into one EAB, the MAX+PLUS II development software can cascade multiple EABs to create the required memory space.

Limited-Transition State Machine

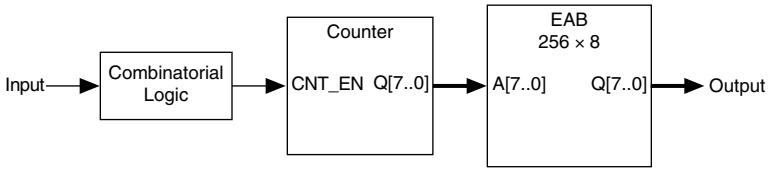
A limited-transition state machine can implement more states in a given amount of logic, but consequently cannot have very complex transitions between states. Figure 9 shows a hold-or-transition state diagram for a limited-transition state machine.

Figure 9. Hold-or-Transition State Diagram



An EAB can control whether a limited-transition state machine remains in the current state or transitions to the next state. First, the inputs to the state machine drive the combinatorial logic implemented in the logic array. The combinatorial logic controls the Count Enable (`CNT_EN`) of the counter. Then, the outputs of the counter drive the EAB inputs, which produce the outputs for that state. For example, an 8-bit counter with one EAB can implement an 8-output, 256-state state machine. See Figure 10.

Figure 10. Implementing a Hold-or-Transition State Machine



If a state machine requires fewer than the maximum number of possible states, the counter can be reset by adding logic. The counter resets after reaching a count value that equals the required number of states. Therefore, if an 8-output state machine with a maximum of 256 states requires only 200 states, adding logic will reset the counter when it reaches 199 states.

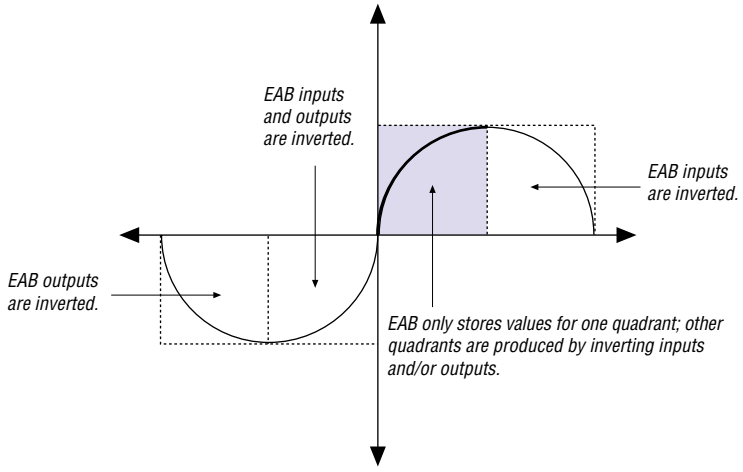
Transcendental Functions

Calculating transcendental functions—such as sine, cosine, and logarithms—with small logic blocks is slow and consumes a large die area. Transcendental functions are non-linear, so they are difficult to compute using algorithms. It is more efficient to implement transcendental functions by looking up the results in large LUTs, which can be implemented with EABs.

When using an EAB to implement a transcendental function, the input drives the address input of the EAB, and the output appears at the data output. Each address location in the EAB stores the result of its input (e.g., the result of the function implemented with input = 10 is stored in address location 10).

Rather than duplicating entries for $+n$ and $-n$ in symmetric functions (e.g., $\cos(+n) = \cos(-n)$, and $\sin(+n) = -\sin(-n)$), transcendental functions use the sign bit to determine whether the output should be inverted. To compute the sine function, for example, the EAB stores the values for one quadrant of the function. Based on the value of the input, the LE computes the results for the other quadrants by determining whether the inputs or outputs of the EAB should be inverted. Figure 11 shows how the values for one quadrant of the sine function can repeat for the rest of the function.

Figure 11. Computing the Sine Function



Computing transcendental functions with an EAB produces a high-resolution result, where resolution is the minimum change in input to change in output. An EAB produces high-resolution results because it can implement functions with high numbers of inputs, whereas LEs cannot easily implement such complex functions. The more entries that can be stored in the EAB, the higher the resolution. One EAB can store 256 8-bit entries. Therefore, an EAB used to calculate a symmetric function effectively has 1,024 8-bit entries because symmetric functions can use each entry in the EAB 4 times, once for each quadrant of the function. For example, computing a sine wave with an EAB produces a resolution of 0.35°.

To increase the precision and resolution on the output of the transcendental function, multiple EABs are used. To increase the precision, one EAB can look up the 8 MSBs while another EAB looks up the 8 LSBs of the result. To increase resolution, two EABs can be used to emulate a 512 × 8 ROM, which provides 2,048 8-bit entries and a resolution of 0.18°.

Waveform Generator

After a sine function has been implemented, the EAB can generate a sine wave. If a counter drives the input of the sine function, the output is a digitized sine wave, and this digital output can be driven to a digital-to-analog converter. A sine wave can be used for various DSP functions.

The EAB can be used to generate waveforms that repeat over time (e.g., sine wave). The waveform generator is implemented with a counter that drives the address input of the EAB, and the waveform output appears on the output of the EAB. Because an EAB can be up to eight bits wide, one EAB can simultaneously generate eight waveforms. Multiple EABs can be cascaded to generate additional waveforms. The waveform can be irregular within its period because it is created with an LUT.

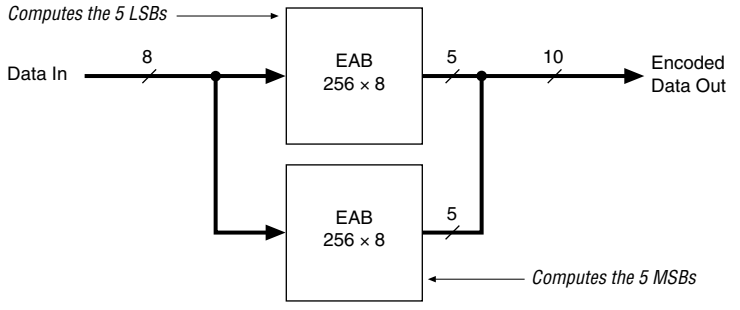
8-Bit-to-10-Bit Encoder

An 8-bit-to-10-bit encoder is used in telecommunications systems. In asynchronous serial telecommunications, the receiving system must synchronize itself while reading the incoming serial data. If the incoming serial data contains a long sequence of 0 or 1 values, the receiving system has trouble synchronizing itself because it cannot detect exactly how many 0s or 1s it has received. To prevent a long sequence of 0s or 1s, the sending system encodes each byte of data into a 10-bit code. There are 768 possible combinations of 10-bit data code that do not correspond to a byte (10-bit data code has 1,024 combinations and a byte has only 256 combinations). If the incoming data code does not correspond to a byte, the receiving system assumes there has been a data transmission error and signals the sending system to re-transmit the data.

Implementing the encoding or decoding circuits consumes many small logic blocks because the relationship between the 8-bit and 10-bit data is non-linear.

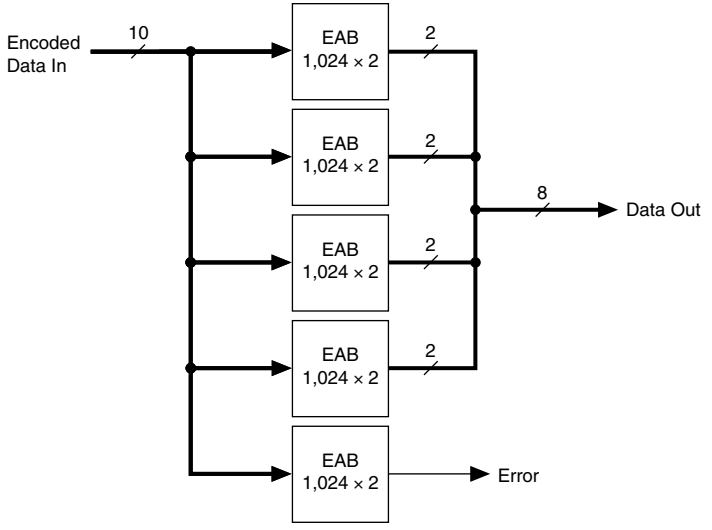
Two 256 × 8 EABs configured as LUTs can be used to encode 8-bit data into 10-bit data. Each EAB is fed by the 8-bit incoming data. One EAB looks up the five LSBs of the output, and the other looks up the five MSBs. A shift register on the output can serialize the outgoing data. See Figure 12.

Figure 12. 8-Bit-to-10-Bit Encoder



The incoming data is decoded using 4 EABs, each configured as 1,024 × 2. The 10-bit encoded data feeds each EAB, which generates two bits of the original data byte. An additional EAB can detect whether one of the 768 illegal 10-bit combinations is received. A shift register can be used on the input to convert incoming serial data to parallel data. Figure 13 shows the implementation of this 10-bit-to-8-bit decoder.

Figure 13. 10-Bit-to-8-Bit Decoder



Other Applications

In addition to the specialized logic applications described in this document, the capabilities of FLEX 10K EABs also allow designers to implement a wide variety of complex combinatorial functions. New combinatorial functions can easily be implemented in the EAB using Altera's MAX+PLUS II development software. Logic options in MAX+PLUS II allow the designer to control the logic synthesis of the design. If a design has combinatorial logic that fits into an EAB, the designer can manually place the logic, or have MAX+PLUS II automatically place the logic in the EAB.

See MAX+PLUS II Help for more information about implementing complex combinatorial functions in EABs.

Conclusion

FLEX 10K devices are the first PLDs to contain embedded arrays. The FLEX 10K embedded arrays, composed of a series of EABs, allow designers to implement complex logic functions in a single level of logic. Using EABs to implement logic functions results in higher device utilization and performance. The flexibility of an EAB makes it well-suited to implement a variety of specialized logic applications and combinatorial functions.

101 Innovation Drive
San Jose, CA 95134
(408) 544-7000
http://www.altera.com
Applications Hotline:
(800) 800-EPLD
Customer Marketing:
(408) 544-7104
Literature Services:
(888) 3-ALTERA