Programmable Memory Blocks Supporting Content-Addressable Memory

Frank Heile, Andrew Leaver, Kerry Veenstra
Altera
101 Innovation Dr
San Jose, CA 95132
USA
1 (408) 544-7000
{frank, aleaver, kerry}@altera.com

ABSTRACT

The Embedded System Block (ESB) of the APEX E programmable logic device family from Altera Corporation includes the capability of implementing content addressable memory (CAM) as well as product term macrocells, ROM, and dual port RAM. In CAM mode each ESB can implement a 32 word CAM with 32 bits per word. In product term mode, each ESB has 16 macrocells built out of 32 product terms with 32 literal inputs. The ability to reconfigure memory blocks in this way represents a new and innovative use of resources in a programmable logic device, requiring creative solutions in both the hardware and software domains. The architecture and features of this Embedded System Block are described.

1. INTRODUCTION

The Embedded System Block (ESB) of the APEX E family of programmable logic devices from Altera Corporation has evolved from the FLEX10K and FLEX10KE Embedded Array Blocks (EAB) [1]. The Embedded Array Block of the FLEX10K is a highly configurable block of RAM that can also be initialized at configuration time to be a ROM block. The block contains 2048 bits that can be configured into RAMs or ROMs of the following sizes: 2048 x 1, 1024 x 2, 512 x 4, 256 x 8. In the FLEX10KE family the EAB has been changed to 4096 bits of dual port RAM with the following configurations: 2048 x 2, 1024 x 4, 512 x 8 and 256 x 16.

In the APEX E family of programmable logic devices, the EAB has been renamed to Embedded System Block because of the additional functionality of CAM mode and product term mode. As a ROM/RAM block, the ESB consists of 2048 bits of dual port memory that can be configured as 2048 x 1, 1024 x 2, 512 x 4, 256 x 8 and 128 x 16. For an overview of the APEX E device family, see the Altera web-site [2] or the APEX E data-sheets [3].

Stansfield and Page [4] previously discussed combining programmable logic devices with CAMs. They proposed a programmable logic device architecture based on logic blocks configurable as any of a 16-bit CAM, 16-bit RAM or a 4-input lookup-table. This architecture also allows for multiple CAM cells to be stitched into a PLA (i.e. product terms).

The CAM mode of an APEX E ESB block has a much greater capability than 16-bit CAMs previously proposed [5][6]. In particular the ESB can directly implement a 32-word CAM with 32 bits per word (see section 4 for definitions and an overview). This CAM mode includes and uses the product term mode of ESB operation. Basically each word of the CAM is implemented in one product term of the ESB. Before we get into a full discussion of how the CAM mode operates and is implemented, we will first describe the product term mode of the ESB.

In product term mode the APEX E ESB can be configured to implement a macrocell similar to the macrocells in the MAX7000 family of CPLDs [7]. A MAX7000 macrocell uses product terms that can be ORed or XORed together to implement logic functions. In particular, the APEX E ESB can be configured to implement up to 16-product term macrocells with 2 product terms each at one extreme, or 1 product term macrocell with 32 product terms at the other extreme. The level of flexibility allowed is that anywhere between 1 and 16 output product term macrocells can be implemented as long as that the total number of product terms does not exceed 32. There are 32 literal inputs available in “true” and “complement” form for all of the product terms in an ESB.

2. PRODUCT TERM MODE

The CAM mode of the APEX E ESB is a superset of its product-term mode. This section reviews the product-term mode of the ESB.

The 10K EAB has 2048 bits of RAM organized as 64 rows and 32 columns. This same organization has been carried over to the APEX E ESB. To implement product term mode for the APEX E ESB, the RAM cell has been modified, some additional circuitry has been added to the row address decoders and the MAX7000 type of macrocell logic has been added to the RAM output block. The modification to the RAM cell was to add a pass gate so that the column bit line will only be pulled low when the cell is programmed to contain a “1” value – the bit line will never be pulled high by a RAM cell. This change is shown in Figure 1. To

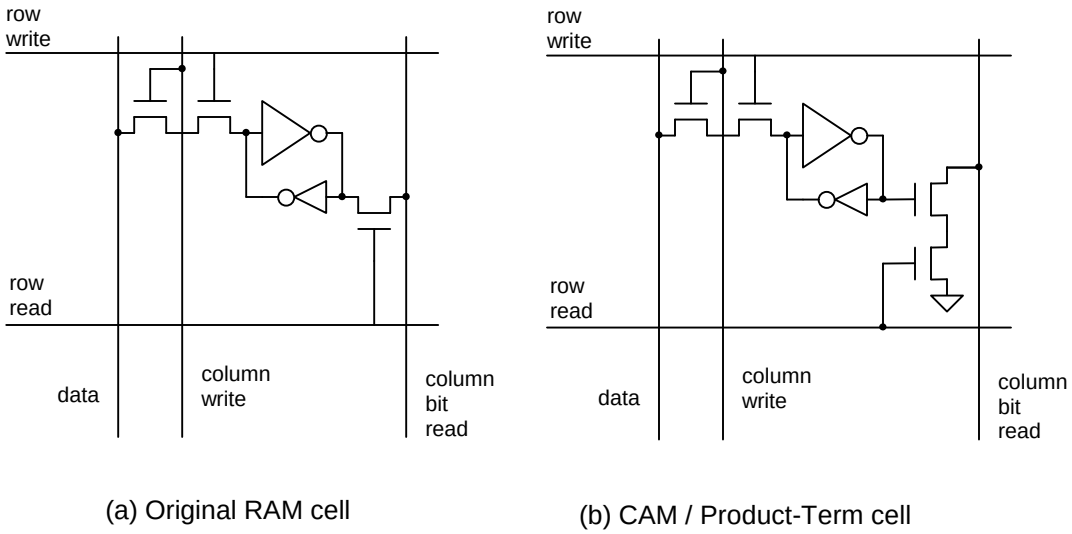

Figure 1 – Memory Cells

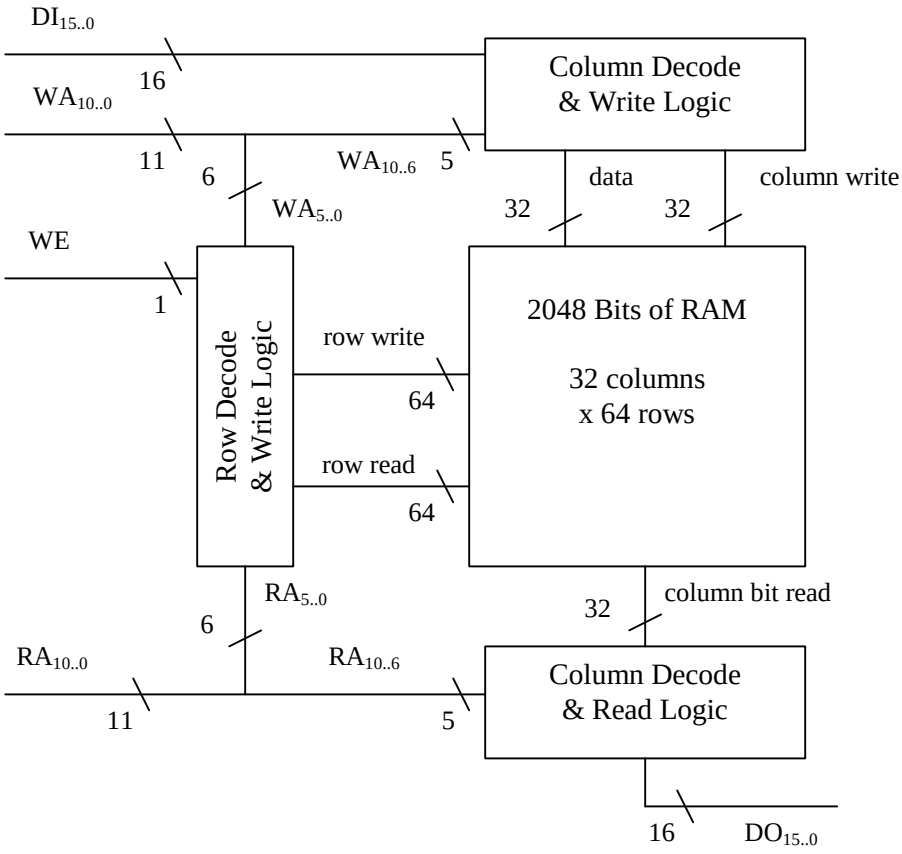

Figure 2 – ESB in RAM mode

make this cell modification work, at the end of the column bit line, instead of a simple buffer, a sense amplifier with a pull-up resistor is used to determine if any of the RAM cells are pulling the column bit line low. Thus if no cells are pulling low, the output will be "1", but if any cell or cells are pulling low, the output will be "0" - thus implementing a logical NOR, which by deMorgan's inversion is equivalent to an AND gate.

A simplified overall block diagram for a APEX E ESB when it is in the RAM or ROM mode of operation is shown in Figure 2. In this figure there are 16 bits of data that can be written into the RAM (DI15..0), 11 bits of write address (WA10..0), 11 bits of read address (RA10..0), a write enable signal (WE) and the data that is output from the RAM (DO15..0). In addition to the connections shown, the read and write column decode blocks need to know the mode of the ESB to correctly enable the 32 column lines used for reading and writing data. The ESB mode will be one of 2048 x 1, 1024 x 2, 512 x 4, 256 x 8 or 128 x 16. The APEX E ESB can also optionally have the inputs and outputs registered. These registers and the associated clocks and other control signals are not shown in Figure 2.

However, when the APEX E ESB is put into product term mode, the simplified block diagram is shown in Figure 3. In this mode the input to the block are the 32 product term literals labeled as D31..0. These 32 data input signals will actually reuse some of the signals that are the inputs the ESB when it is in RAM mode. In particular, the 16 bit data input bus, the 11 bit read address bus and the low order 5 bits of the write address bus will be used as the 32 literal input signals for the ESB in product term mode. The outputs from the RAM block will be the 32 signals labeled as "column bit read" which are then the inputs to the sense amps and macrocells.

Figure 4 shows the macrocell logic for the APEX E ESB. Since there are 16 output drivers available for the ESB block when it is in RAM mode and since there were 32 product terms available it has been an easy decision to create 16 macrocells with 2 product terms per macrocell. The APEX E ESB macrocell is very similar to the MAX7000 macrocell except that there are only 2 product terms instead of the 5 product terms per macrocell with MAX7000. Another difference is that the clock and other secondary signals for the flip-flop of the macrocell are selected from an ESB wide set of signals, which is similar to the way that the secondary signals for flip-flops in the regular 4-LUT LABs are selected. In particular the ESB has 2 clocks and 2 clears and each flip-flop can independently choose from those clocks and clears. Of course each macrocell can also choose to bypass the register and instead output a combinatorial signal.

The product term selection matrix allows each macrocell to use anywhere from 2 to 32 product terms. When the 3rd through 32nd product terms are used, they will be borrowed from the other macrocells in the ESB and are called parallel expanders. The ESB does not implement the shared logic expander capability that is available in the MAX7000 devices. As a substitute for this shared logic expander feature, one of the product terms in each macrocell can be inverted. This allows a logic function, where some number of the product terms consist of single literals, to have all of these single literal product terms implemented in just one inverted product term.

3. CONTENT ADDRESSABLE MEMORY MODE

3.1 Background

Before we get into a discussion of the ESB implementation of content addressable memories (CAMs), we will first describe what a CAM is in general. For an ordinary RAM memory, a DATA word is stored at an ADDRESS during a write cycle. During a read cycle, when that ADDRESS is again presented to the RAM and the DATA word that was stored is retrieved. A CAM is usually described as a kind of inverted memory because in the CAM read cycle the DATA word is input to the CAM, and the CAM then outputs the ADDRESS where that data is stored.

When described in these terms, it may not be clear why a CAM is any different than a RAM. After all, the labels of ADDRESS and DATA are somewhat arbitrary and if the labels are simply reversed, then a CAM sounds exactly like a RAM. For example, if it is guaranteed that there is a one-to-one relationship of DATA and ADDRESS (i.e. the same data is not "stored" in multiple addresses), then a CAM can be emulated by a RAM where the number of address bits is equal to the number of bits in "DATA" and the word size is the number of bits of "ADDRESS". Then when the RAM is presented with the "DATA" bits on the RAM address bus, the RAM (emulating a CAM) would be able to output the "ADDRESS" on the RAM output bus. The problem with this concept is that the number of bits in the "DATA" word may be large, and thus this implementation may require a huge equivalent RAM. As an example, let's consider a CAM where there are 32 words with 32 bits per word. This would mean that "DATA" is a 32 bit quantity, but there are only 5 bits of "ADDRESS". To implement this CAM in an ordinary RAM we would require 2^32 (= 4,294,967,296) 5-bit words of RAM. However, in this example, only 32 of the 2^32 words would contain the "ADDRESS" (5 bit) contents. In fact a CAM of that size can be implemented in one APEX E ESB using only the 2048 bits of RAM in the ESB. Thus in a sense CAMs allow a very large very sparse RAM to be implemented efficiently.

However, CAMs can be even more effective than just a very large sparse RAM. This is because when the "DATA" is stored in the CAM, some of the bits of the data can be specified as being don't care bits. We can extend the example used above by assuming that, for example, 10 bits of the 32 bit "DATA" quantity that are specified as don't care bits. Then when the "DATA","ADDRESS" pair is written in to the sparse RAM, the "ADDRESS" word would actually have to be written into 2^10 = 1024 different locations to account for these 10 don't care bits in "DATA". Thus a write cycle of this "DATA" and "ADDRESS" would require 1024 RAM writes to implement these don't care bits and would thus be very slow.

3.2 Read Implementation

With the addition of a 32-to-5 encoder in the output block, the APEX E ESB in product term mode can implement a 32 word CAM with a 32-bit "DATA" word. The output of the ESB will be a 5-bit "ADDRESS" to indicate which product term is true, and a one-bit "Match" flag to indicate if any product term is true. This mode of operation for the CAM assumes that there will only be a single match – i.e. at most a single product term will be true for whatever data is input. The use of a product term to recognize the "DATA" word allows some of the bits in each word to be

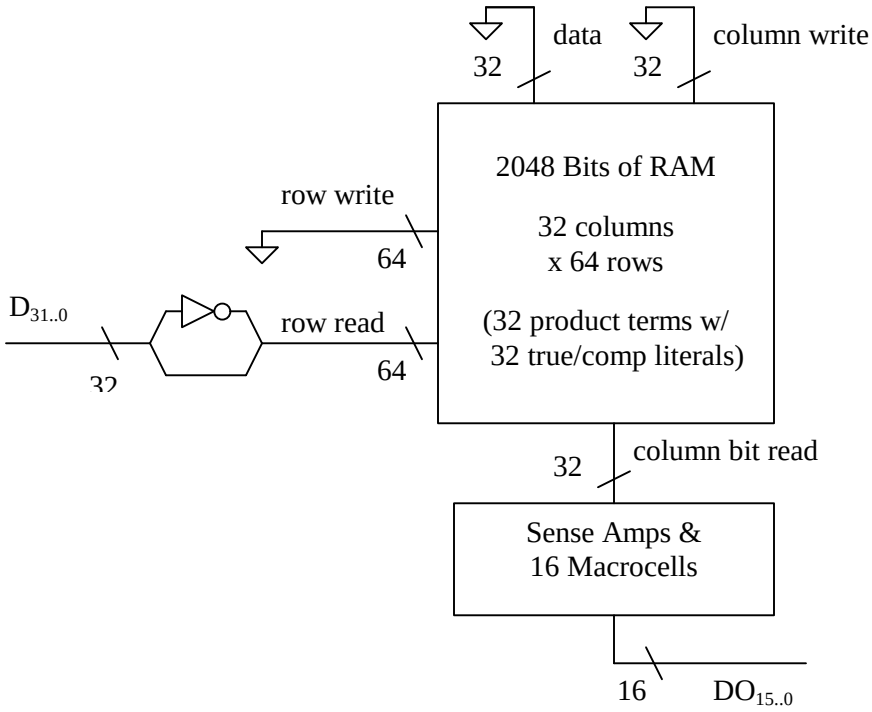

Figure 3 – ESB in Product Term Mode

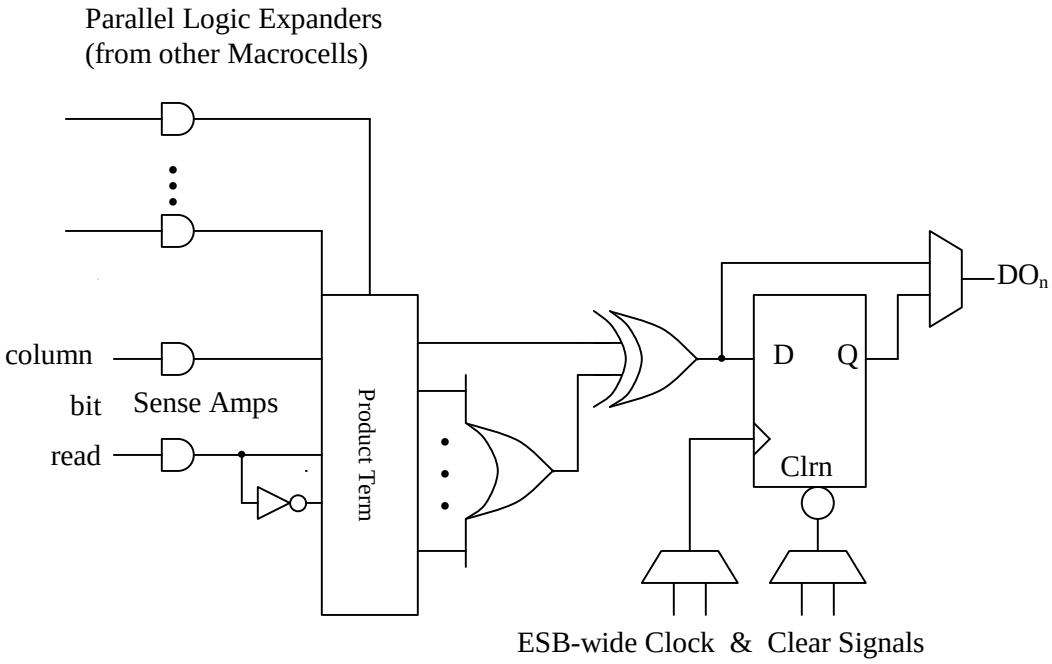

Figure 4 – ESB 2 Product Term Macrocell

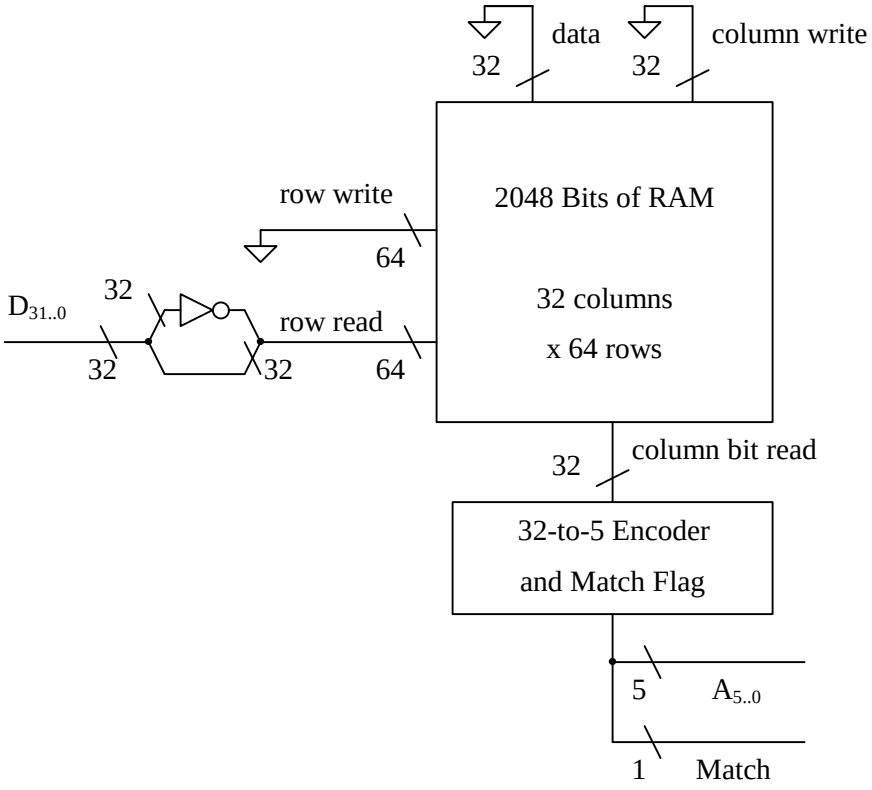

Figure 5 – ESB in CAM Read-Only Mode

specified as don't care bits – in fact, each word can potentially have different sets of don't care bits. This use of an ESB where we assume the product terms are already "programmed" implements a read-only CAM and is shown in Figure 5. Notice the similarity to Figure 3, the only difference being in the change to the output block from macrocells to an encoder.

So we now have a Read-Only CAM implementation by just adding the encoding logic to the ESB product term mode. If we can now figure out how to write to the product term RAM bits, we will have a complete Read-Write CAM.

3.3 Write Implementation

The first idea explored for writing to the product term mode RAM bits would be to use the normal RAM mode write circuitry. However there are a couple of problems with this approach. First of, depending on the RAM configuration chosen, we will write 1, 2, 4, 8 or 16 bits at the same time. However when multiple bits are being written simultaneously, they are be written to multiple columns of the RAM bit array. Unfortunately a product terms bits are all in a single column of the array. Thus the only way to write a column of bits with any of the normal RAM mode would be to write the data in the 1 bit mode. Since there are 64 bits in a column for a single product term, this will require 64 write cycles per CAM word written – not a very efficient way to write a word. The other problem with this approach is that the total number of signals required will be more than the total number of signals we planned to be available to an ESB. For example, to write a bit in the 2048 by 1 mode requires the following: 1 signal for the data, 11 signal for the address and 1 write enable signal. These 13 signals plus the 32 signals needed for the read mode exceeded the planned number of signals that would be available in an ESB. We could still make this work by reusing some of the read mode signals for the write mode, but it would have added complications to the logic outside the ESB. It would also have been challenging to make sure that the 64 bits of data were being written into the correct RAM bits that would depend on exactly how the 32 bits of data were routed into the ESB.

The method we used for implementing the CAM write mode is much more elegent and efficient than this first idea. The basic requirement is to write as many bits as possible in a single column in order to make the CAM write mode as fast as possible. Now the normal row decoder that generates the 64 "row write" signals (see Figures 1 and 2) will only allow one of the 64 signals to be true at the same time. So to write more than one bit at a time we need to enable multiple row write signals to be true at the same time. The solution has been to use the true and complement values of the 32 literals (D31..0 of Figures 3 or 5) that are used during the CAM read mode to control the CAM write mode. We do this by connecting (through some gates) the row read signals to the row write signals. The extra logic enables us to invert all the signals or to force all 64 of the row write signals to "0" when we are not writing. Since 32 of the 64 row read, and now row write signals, are true at any given time, we will be writing to exactly 32 bits at a time in the column where writing is enabled.

The final solution that has been adopted for the CAM read/write mode is shown in Figure 6. The additional logic that has been added to Figure 6 (compared to Figures 2, 3 or 5) is the additional signal InvD, the XOR and AND gates that are used to feed the 64 "row read" lines onto the 64 "row write" lines. The block labeled "Column Decode & Write Logic" is the same as the corresponding block in Figure 2, except that it is always in the x1 write mode. That is why there is only a single data input bit (DI0) to this block shown in Figure 6 – because we are only writing one column at a time. Remember that we will be writing data into multiple bits (32) in the chosen column (specified by the 5 bits labeled WA10..6 and the value of this DI0 bit will determine whether we are writing a "1" or a "0" to those multiple RAM bits. Thus in general, to write an entire column will require at least two write cycles – one to write the "1" bits and another cycle to write the "0" bits. (We will show later that a third cycle may be required when "DATA" words with don't care bits are to be written).

To explain how this works, let's first review the details of how product terms work.

3.3.1 How Product Terms Work

During read mode each of the 32 input literals (D31..0) are presented in both the "true" and "complement" form on the 64 "row read" lines of the RAM block. Thus there are two RAM bits that will affect how a given literal input influences the output of the product term. Calling these two bits the "true RAM bit" and the "complement RAM bit" the following table shows the effect on the product term output given the literal input:

Literal Value	True RAM bit	Comp. RAM bit	Product Term Output	Product Term Meaning
"1"	"0"	"1"	"1"*	"Recognized"
"0"	"0"	"1"	"0"	Not recognized
"1"	"1"	"0"	"0"	Not recognized
"0"	"1"	"0"	"1"*	"Recognized"
X	"0"	"0"	"1"*	Don't-care literal bit
X	"1"	"1"	"0"	Disabled product term

* = if no other literal is pulling low ("0")

3.3.2 Writing Data without Don't Cares

So, for now if we consider the case where there are no "don't care bits", we need to either write "1, 0" or "0, 1" in the "true, complement" RAM bit locations for a given literal. The pattern written ("0, 1" or "1, 0") would depend on whether we want to recognize a literal value of "0" or "1" respectively. Thus the 64 bits of a product term column for which all 32 bits of the literal contribute to the output will have exactly 32 "1" bits and 32 "0" bits in that column. The waveforms for the signals DI0, InvD and WE are shown in Figure 7. The basic idea is that while InvD and DI0 is "0", WE is pulsed so that a "0" value is written to:

- all "true RAM bits" for all literals where the literal value is "1", and
- all "complement RAM bits" for all literals where the literal value is "0".

Then in the second cycle, InvD is set to "1" while DI0 is set to "1" so that the "1" values are written to

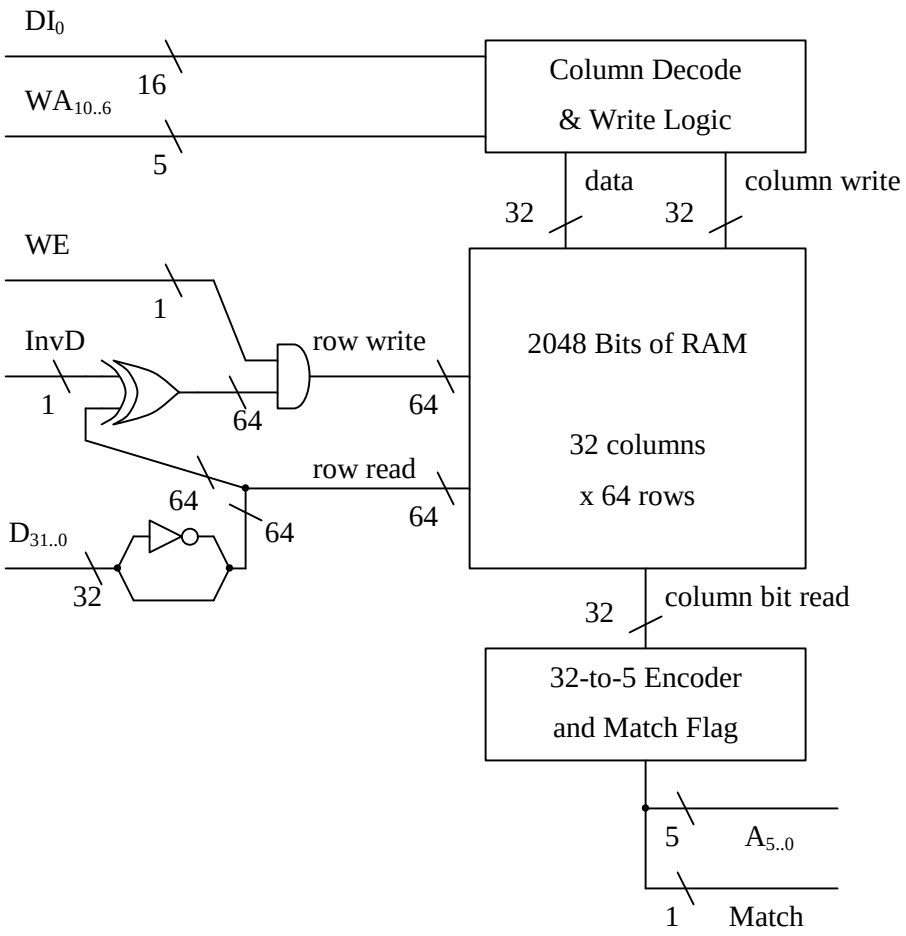

Figure 6 – ESB in CAM Read-Write Mode

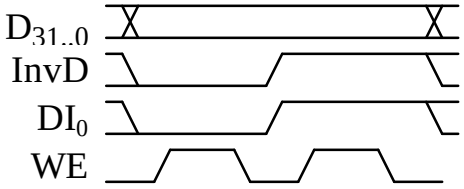

Figure 7 –
ESB CAM write cycle if there are no don't care bits (two cycles)

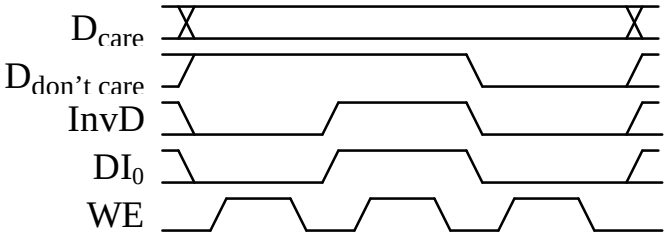

Figure 8 –
ESB CAM write cycle if there are don't care bits (3 cycles)

- all “complement RAM bits” for all literals where the literal value is “1”, and
- all “true RAM bits” for all literals where the literal value is “0”.

Note that in the second cycle a “1” bit value is written in the opposite set of RAM bits compared to the set of bits that were written as a “0” value in the first cycle. Thus at the end of these two memory write cycles, the “true” and “complement” bits for each literal will be set to either “0, 1” or to “1, 0” respectively.

3.3.3 Writing Data with Don’t Cares

The case where some of the literal bits are to be treated as don’t cares is only slightly more complicated. First of all, we need to set all the literals that are to be “don’t cares” to “1” and complete the first two memory write cycles as discussed previously. Then for a third memory write cycle, set the literals which are to be “don’t cares” to “0” and perform the first write cycle again. This is shown in Figure 8. What happens is that at the end of the first two cycles the “true, complement” bits for these don’t care literals will be set to “0, 1”. Then when the third memory cycle comes along a “0” value is written into the “complement” bit, thus giving a final pattern of “0, 0”. This is exactly the value need for a don’t care literal. No matter what the value of the literal, neither of the two RAM bits will ever cause the column bit line to be pulled to “0”. For a product term with no don’t care bits, there are exactly 32 “0” bits and 32 “1” bits in a column – and since each write cycle will each write exactly 32 “0” or “1” bits, that is why two write cycles is sufficient for this case. However when a product term has some don’t care bits, there will be more than 32 “0” bits in a column – and thus a third write cycle is required since the third cycle must overwrite some of the “1” bits written in the first two cycles.

So we have achieved the goal of having a fast and flexible method for writing a single word of DATA (i.e. a single 64-bit product term) in the CAM mode of the ESB. In the case where there are no don’t care bits, the write can be completed in two memory cycles with only a very minimal state machine implemented in logic outside of the ESB to control the DI0, InvD and WE signals. No logic is required on the signals representing the 32-bit word that is being written into the CAM. And finally the correct bits will always automatically be written independently of the way in which the 32-bit data word has been routed into the ESB. In other words, reordering the bits in the 32 bit data word will have no effect on the logic needed to implement the CAM write since the same ordering will apply to both the CAM read and write mode.

In the case where there are some don’t care bits that need to be written into the CAM, it will take slightly more external logic in the state machine since there are three memory write cycles per CAM write instead of two. There will also need to be some logic on whichever data bits could potentially be a don’t care bit so that the proper values can be set on those bits during the 3 write cycles. The exact set of don’t care bits for each word that is written can be determined independently at run time, but one extra 4-LUT logic cell will be needed on each bit that can potentially be a don’t care bit.

Note that if less than 32 bits are desired, it is not necessary to consider the extra unused bits as don’t care bits – instead they can just always be set to “0” at all times and they will then have no effect.

3.4 Handling Multiple Simultaneous Matches

In terms of the CAM mode outputs, we noted that there is a 5 bit encoded output plus a match flag. These are appropriate and useful only if the logic designer knows that there will never be a multiple match situation during the CAM operation. If either the same value is written into two or more addresses, or if the don’t care bits for two words allowed some value to match both words, then two or more product terms can potentially go to a “1” state at the same time. The resulting encoded output will in this case be the “OR” of the two correct addresses – obviously an incorrect answer.

If the logic designer needs to handle this multiple match case, then the encoded outputs should be turned off and instead the individual product terms should be output thru the product term macrocell mode to let external logic handle the multiple match case. With the encoded outputs turned off, the normal MAX7000-like macrocell logic can be used in the ESB. One problem with this approach is that there are only 16 macrocell outputs available per ESB, but there are 32 product terms in an ESB. One solution to this problem is to use only 16 of the 32 product terms, i.e. use only half of the ESB capability. The other solution would use up 1 of the 32 input bits and require that two memory read cycles are used to get all 32 product term values out of the ESB. This is done by programming, say, the even product terms so that if this 32nd input is a “1”, they will be recognized, and program the odd product terms so that they are recognized when this 32nd input is a “0”. Thus, by toggling this 32nd input between two read cycles, all 32 product terms can be read out and stored in 32 external flip-flops where the required multiple match logic can then be performed. So basically, if the user requires multiple match capability, the choice is to either use half the product terms at full speed, or use all the product terms at half speed.

4. PUTTING IT ALL TOGETHER

In Figures 1 through 6 we have shown the major architectural features of the APEX E ESB block. What are not shown are many of the important details. For example, in most cases, the RAM and/or CAM ESB has been treated as if it was an asynchronous device. In actuality, we expect most applications will probably use the ESB in a synchronous manner. For example, in the ESB all of the read and write addresses, the data to be written and the various control signals can all be synchronously registered to one or another of the two clocks available per ESB, or they can be asynchronous. We expect the synchronous mode to be used more often because the timing requirements are much easier for the synchronous design. In addition the synchronous design technique is ideal for pipelined designs.

An example of another detail left out is that various muxes have been eliminated for clarity. These muxes will typically choose between using one or another signal depending on the ESB mode chosen. For example, if you examine Figures 2, 3, 5 and 6 you would see that we need at least the following muxes which have not been shown in any Figure:

- On the 64 “row read” and “row write” lines a 2-to-1 mux is required to choose between the signals generated by the

"Row Decode and Write Logic" (Figure 2) and the product term/CAM read/write row logic (Figures 3, 5 and 6).

- In the vicinity of the 16 ESB outputs some 2-to-1 and 3-to-1 muxes would be needed for each output to either output the RAM data read (Figure 2), the macrocell logic (Figure 3) or the encoded CAM outputs (Figures 5 & 6). In this case, only 6 outputs need a 3-to-1 mux, the other 10 just need a 2-to-1 mux.

5. CONCLUSIONS

In this paper we have described a novel method for re-using memory blocks in a programmable logic device to implement either content-addressable memories or product term logic. Our ideas have been implemented in the embedded system block of the APEX E family of programmable logic devices from Altera.

With product term support we are able to better utilize die area by trading off the logic and memory requirements specific to individual designs, and can accommodate wide-input functions more suitable to product term implementation than they would be in a lookup table. With CAM support, we can more efficiently implement specialized forms of memory, which would otherwise be difficult to achieve.

6. REFERENCES

[1] Altera Corporation, "Device Data Book", 1999.

[2] http://www.altera.com/

[3] Altera Corporation, "APEX 20K Programmable Logic Device FAmily, Data Sheet", 1999.

[4] A. Stansfield and I. Page, "The Design of a New FPGA Architecture", Proceedings of the 5th International Workshop on Field-Programmable Logic and Applications, FPL 95, Oxford University, August 1995.

[5] S. Wilton, J. Rose and Z. Vranesic, "Memory-to-Memory Connection Structures in FPGAs with Embedded Memory Arrays." Proceedings of the 5th International Symposium on FPGAs, FPGA 97, Feb 1997. (Submitted to IEEE Trans. VLSI).

[6] S. Wilton, J. Rose and Z. Vranesic, " Memory/Logic Interconnect Flexibility in FPGAs with Large Embedded Memory Arrays," in CICC 96, the IEEE Custom Integrated Circuits Conf., San Diego, CA, May 1996, pp. 144-147.

[7] F. Heile, A. Leaver, "Hybrid Product Term and LUT Based Architecture Using Embedded Memory Blocks", Proc. ACM 7th International Symposium on FPGAs, FPGA 99, Monterey, CA., Feb. 1999.