

# An Embedded Flexible Content-Addressable Memory Core for Inclusion in a Field-Programmable Gate Array

Steven J.E. Wilton  
University of British Columbia  
Vancouver, B.C., Canada  
steve@ece.ubc.ca

Christopher W. Jones  
Greenville, SC  
Chriscjones@aol.com

Julien Lamoureux  
University of British Columbia  
Vancouver, B.C., Canada  
julienl@ece.ubc.ca

## Abstract

*This paper describes a novel architecture for a content-addressable memory core which is intended to be included in a Field-Programmable Gate Array or an Embedded Programmable Logic Core. Compared to a standard content-addressable memory, our architecture allows for arbitrary data and tag widths; this flexibility is vital in FPGA applications, since the size of the data and tag fields are not known when the FPGA is manufactured.*

## 1.0 Introduction

Recent years have seen impressive improvements in the achievable density of integrated circuits. Accompanying these improvements, however, is a dramatic increase in the cost of fabricating an integrated circuit. No matter how seamless the integrated circuit design flow is made, and no matter how careful a designer is, there will always be some chips that are designed, manufactured, and then deemed unsuitable. This may be due to design errors not detected by simulation or it may be due to a change in requirements. Coupled with the dramatic increase in mask costs and design costs, these potential re-spins may make the design of a state-of-the-art integrated circuit impractical for all but the highest-volume designs.

Programmable logic can help sidestep much of these costs. In low to medium-volume applications which do not have aggressive speed and power requirements, stand-alone Field-Programmable Gate Arrays (FPGAs) can eliminate the need for fabrication entirely [1-3]. FPGAs can be configured to implement virtually any digital circuit after fabrication, however, FPGA implementations are typically between 3 and 10 times less dense and between 3 and 10 times slower than the same design implemented on a custom chip. For applications in which this overhead is not acceptable, a custom chip can be created, and a programmable logic core can be embedded [4] to provide post-fabrication flexibility just to those parts of the chip that are expected to change. Programmable logic cores have a similar architecture to stand-alone Field-Programmable Gate Arrays, however, they do not need power-hungry and area-hungry input/output circuitry, since they can be wired directly to the hardwired circuitry on the chip.

Regardless of whether a stand-alone FPGA or an embedded programmable logic core is used, the architecture of the programmable logic fabric has a significant effect on the efficiency

(density, speed, and power) of the device. Recent FPGA architectures are vastly different than the architectures of five years ago. One of the most significant enhancements to FPGA architectures in recent years has been the introduction of fixed-function cores, or sub-blocks. These cores include memory components, processors, DSP blocks, and embedded multipliers [1-3]. Although any of these functions can be implemented using general-purpose programmable logic, embedding hardwired blocks within a programmable logic fabric provides significant density, speed, and power reductions for circuits that employ these functions.

Most of the embedded functions that FPGA vendors have chosen to incorporate in their parts are aimed at signal processing and communications applications. One of the most important types of cores for these applications is an embedded memory [5]. Unlike embedded memory in a fixed-function chip, the data width and other access requirements in FPGA embedded memories are not known when the chip is fabricated. Thus, the primary requirement of these embedded memory architectures is that they be flexible enough to cater to as many different applications as possible.

A type of memory commonly used in many types of communications-related circuits is a content-addressable memory (CAM). In a content-addressable memory, each stored word consists of two sets of bits: tag bits and data bits. During a read operation, a set of tag bits is supplied to the memory. These tag bits are compared to the tag bits stored in every memory location simultaneously. If the supplied tag bits match the tag bits stored in any of the memory locations, the memory array indicates that a match has occurred, and the matching word can be read.

Content-addressable memory have been included in the APEX 20K parts from Altera [6]. In these devices, each Embedded System Block (ESB) can optionally be configured as a CAM with 32 words of 32 bits each. Combining these blocks to implement larger CAMs is troublesome. The number of data bits per word can be expanded by connecting multiple blocks in parallel, and replicating the tag bits within each block. Since the tag bits must be replicated, this leads to an inefficient implementation of large CAMs. Furthermore, there is no simple way to implement CAMs with wider tag widths.

In this paper, we present a novel architecture for a content-addressable memory that provides arbitrary tag and data widths. The intention is that this block would be incorporated into a Field-Programmable Gate Array or a Programmable Logic Core, however, it can be used whenever post-fabrication flexibility of the CAM is desired. Section 2 describes the overall organization of the CAM, Section 3 describes the architecture, Section 4 describes how the CAM can be read and written, and finally, Section 5 presents some overhead estimates for our architecture. A longer description of this architecture appears in U.S. Patent 6,622,204 [7].

---

<sup>1</sup> This work was performed when both authors were at Cypress Semiconductor, San Jose, California. The subject of this paper is based on the U.S. Patent: C.J. Jones, S.J.E. Wilton, "Content-addressable Memory with Cascaded Match, Read and Write Logic in a Programmable Logic Device", U.S. Patent 6,622,204. Issued Sept. 16, 2003, assigned to Cypress Semiconductor Corporation.

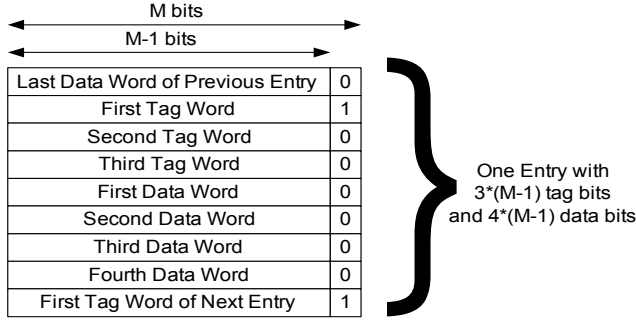


Figure 1: Memory Map showing one entry in the CAM

## 2.0 CAM Storage Organization

In a typical CAM, each memory word is divided into two parts: a tag field and an address field. Each tag field is associated with one comparator. Each comparator compares the associated tag with the input tag bits, and if a match occurs, the corresponding data bits are driven out of the memory. Although this is fast, it is not suitable for applications with a very wide tag or data width. Wide tags lead to large comparators, which are area inefficient, power hungry, and often slow. In addition, the tag and data widths must be fixed at design time. As described above, one of the primary requirements of an FPGA CAM block is that it be flexible enough to take on many different tag and data word widths.

In our architecture, each memory word is of a fixed size. However, multiple memory words can be used to store each tag field and each data field. Consider the memory map of Figure 1. In this figure, each memory word is  $M$  bits wide (a typical value of  $M$  might be 9 or 11). In the example of Figure 1, each tag field is  $3(M-1)$  bits long, and is stored in three consecutive memory cells. Each data field is  $4(M-1)$  bits long, and is stored in four consecutive memory cells. One bit of each word is used to indicate the start of a new tag entry. Clearly, with this sort of structure, any arbitrary data and tag width can be created.

## 3.0 Architecture

Figure 2 shows the overall architecture of our CAM. The core consists of  $N$  words, each  $M$  bits wide. Each word is accompanied by match logic and a small processor. Each word is driven by  $2M$  data-in lines, 5 instruction lines, and drives  $M$  data-out lines. The user uses the  $2M$  data lines to specify the match condition; each bit is associated with two data-in lines. Two lines per bit allows the user to specify a “don’t care” condition for each bit. Each word  $i$  provides a  $gmatch_i$  signal to the global match logic which indicates whether this word matches the tag input. The global match logic block arbitrates between these  $gmatch$  signals, and produces an output  $single\_match$  which is asserted if exactly one word matches the tag, and an output  $multiple\_match$  which is asserted if more than one word matches the supplied tag.

Figure 3 shows the structure of each memory word and its associated processor. As explained in Section 2, a tag word or data word may span more than one memory word; the purpose of the processor (labeled “Match Control” in the diagram) is to combine

the match results from consecutive memory words. If word  $i$  matches, the local comparator asserts  $local\_match_i$ . The match control logic then combines this result with the match result from either the word above or the word below.

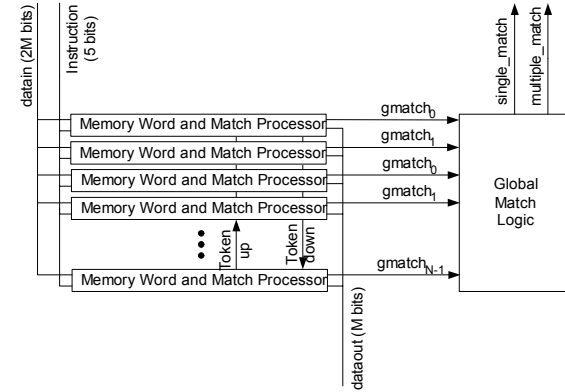


Figure 2: Overall CAM Architecture

The instructions performed by the processor are shown in Table 1. In this table, the letter D is used to indicate “direction” (0=up, 1=down). Note that all processors receive the same instruction. The processor itself is a simple logic circuit that performs the functions in Table 2. Section 4 will show how these operations can be used to match, read, and write data.

## 4.0 Memory Accesses

Using the instructions in Table 1, a variety of read, match, and write operations can be performed. In this section we describe four common operations.

### a) Reading when the tag and data bits fit in one word

In the case when the combined sizes of the tag and data fields is less than one memory word (less than  $M$ ), the process outlined in Table 3 can be performed. In the first step, the user places the match bits on the data input lines. There are two data input lines per memory word bit; a value of “00” means the corresponding bit must be 0, a value of “01” means the corresponding bit must be 1, and a value of “10” means the corresponding bit doesn’t matter. Typically, the inputs for bits that are act as tag bits will be set to “00” or “01”, and the inputs for bits that are to act as data bits will be set to “10”.

Instruction	Encoding
Match	00000
Cascade_match	0101D
Shift then read	1001D
Read then shift	1011D
Read	10000
Shift then write	1101D
Write then shift	1111D
Write	11000

Table 1: Instruction Set for each processor

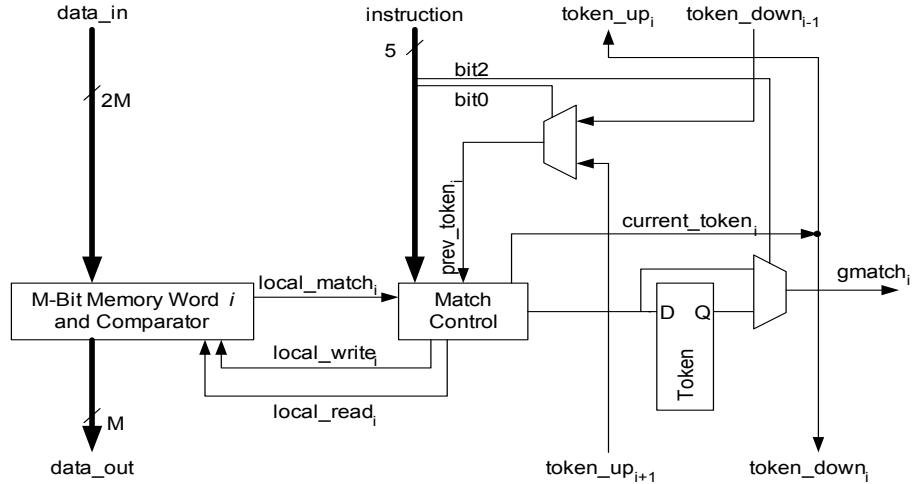


Figure 3: Structure of each memory word and its associated processor

After the read operation, all matching words will drive the data output lines. In general, more than one word may match, however, the output is only meaningful if exactly one word matches. Thus, the memory contains a *single\_match* output and a *multiple\_match* output, allowing the external access circuitry to interpret the results on the data output lines correctly.

Instruction	Operation
MATCH	new_token = local_match local_write = 0 local_read = 0
CASCADE_MATCH	new_token = prev_token AND local_match local_read = 0 local_write = 0
SHIFT_THEN_READ	new_token = Prev_token local_read = prev_token local_write = 0
READ_THEN_SHIFT	new_token = prev_token local_read = current_token local_write = 0
READ	new_token = current_token local_read = current_token local_write = 0
SHIFT_THEN_WRITE	new_token = prev_token local_read = 0 local_write = prev_token
WRITE_THEN_SHIFT	new_token = prev_token local_read = 0 local_write = current_token
WRITE	new_token = current_token local_read = 0 local_write = current_token

Table 2: Processor Operation

*b) Writing when the tag and data bits fit into one word*

The write operation is performed in the same way as the read operation, except that the WRITE instruction is used in Step 2 of Table 3 rather than the READ operation.

*c) Reading when the tag and data comprise multiple words*

When the data and/or tag fields comprise more than one word, sequential accesses are required. Table 4 shows the sequence of operations required when the tag field comprises three words and the data field comprises four words (as in the example of Figure 1).

In the first step, the user places the first M-1 match bits on the data input lines. The input bits corresponding to the tag-start field (the LSB of each word in Figure 1) are set to "01" meaning only words that contain the first word of a tag field will possibly indicate a match. As a result of the MATCH operation, a 1 is written into the token register (see Figure 3) corresponding to each word that matches.

In the second and third step, the remaining match bits are presented, and the instruction input is set to CASCADE\_MATCH. As shown in Table 2, this command is similar to MATCH, except that a match occurs only if the current word as well as all previous words match. At the end of the third step, only those entries which matched in all three steps will contain a 1 in the token register.

Step	Instruction	Operation and Result
1.	MATCH	The user places the match bits on the data input lines. All words are compared to the match bits
2.	READ	All matching words are driven out of the memory

Table 3: Operations to perform a read when tag and data words fit in one memory word

After a match has been found, the next four steps read the corresponding data bits. This is done using the SHIFT\_THEN\_READ instruction, which shifts the token bit one word. After the shift, all words with a corresponding token bit of 1 are then sent out of the memory on the data output lines. In our example, this would be repeated four times, each time, an additional M-1 bits being read.

*d) Writing when the tag and data comprise multiple words*

A match followed by a write can be accomplished using the same set of instructions as in Table 4, except that each SHIFT\_THEN\_READ is replaced by a SHIFT\_THEN\_WRITE. Alternately, a match/read/write cycle can be performed using the sequence of instructions in Table 5. In this case, the match and read are performed as before, however, after being read, the word is written, this time starting from the last word. During the write operation, the token bit is shifted “up” rather than “down”; this is done by setting the least significant bit (D in Table 1) appropriately during the write operations.

Step	Instruction	Operation and Result
1.	MATCH	The first M-1 bits of the tag are placed on the data input lines. The token register corresponding to each matching word is set.
2.	CASCADE_MATCH	The next M-1 bits of the tag are placed on the data input lines. The token register corresponding to each matching word is set if the previous word also matched.
3.	CASCADE_MATCH	The final M-1 bits of the tag are placed on the data input lines. By the end of this step, the token register is 1 for all words whose entire tag field matches.
4.	SHIFT_THEN_READ	The token bit is shifted to the next word which contains the first M-1 bits of the data. These M-1 bits are driven out of the memory.
5.	SHIFT_THEN_READ	The next M-1 bits of the data are read out of the memory.
6.	SHIFT_THEN_READ	The next M-1 bits of the data are read out of the memory.
7.	SHIFT_THEN_READ	The final M-1 bits of the data are read out of the memory.

Table 4: Operations to perform a read when tag and data words require multiple memory words

Step	Instruction	Operation and Result
1-7	As in Table 4	
8.	WRITE_THEN_SHIFT	The last M-1 bits of the data are written, and the token is shifted to the previous word.
9.	WRITE_THEN_SHIFT	The next M-1 bits of the data are written, and the token is shifted.
10.	WRITE_THEN_SHIFT	The next M-1 bits of the data are written, and the token is shifted.
11.	WRITE_THEN_SHIFT	The first M-1 bits of the data are written, and the token is shifted.

Table 5: Additional operations to perform match/read/write

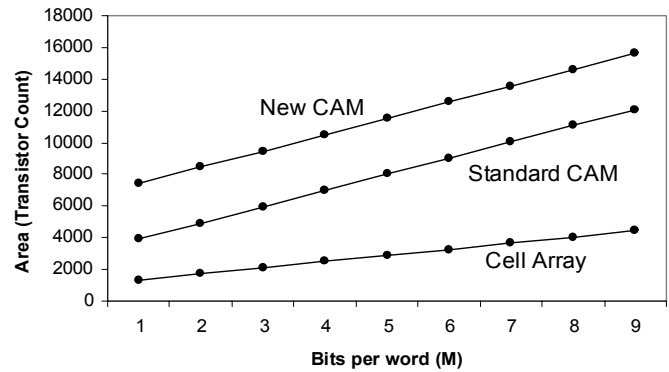


Figure 4: Area overhead as a function of M (bits per word)

**5.0 Area overhead estimation**

To estimate the area overhead inherent in our architecture, we have estimated the number of transistors in our proposed CAM as well as the number of transistors in a traditional CAM, as a function of M, the number of bits in each word. Figure 4 shows the results. The overhead ranges from 30% to 90%. In gathering the data in Figure 4, 32 memory words were assumed (N=32), although the relative overhead numbers are the same regardless of N. Although this overhead is significant, it compares favourably with the overhead of standard FPGAs compared to ASICs (which can be as much as 10x). For reference, the graph also shows an estimate of the number of transistors in the memory cells themselves.

**6.0 Conclusions**

In this paper we have described a novel architecture for a content-addressable memory core which is intended to be included in a Field-Programmable Gate Array or an Embedded Programmable Logic Core. The primary requirement of such a core is that it be flexible, since the application for which the core will be used is not known when the FPGA is manufactured. Our architecture allows arbitrary tag and data width. The overhead ranges from 30% to 90% compared to a traditional CAM architecture.

**References:**

- [1] Xilinx Inc., “Virtex-II Pro Detailed Functional Description”, v2.8, Sept 2003.
- [2] Altera Corp, “Stratix Device Family Datasheet”, v2.0, Jul. 2003.
- [3] Actel, “ProASIC<sup>PLUS</sup> Flash Family FPGAs”, v3.2, Aug. 2003.
- [4] S.J.E. Wilton, R. Saleh, “Programmable Logic IP Cores in SoC Design: Opportunities and Challenges”, in the *IEEE Custom Integrated Circuits Conference*, San Diego, CA, May 2001, pp. 63-66.
- [5] S.J.E. Wilton, “FPGA Embedded Memory Architectures: Recent Research Results”, in *IEEE Pacific Rim Conference on Communications, Computers and Signal Processing*, Aug 1999.
- [6] Altera Corp, “APEX 20K Programmable Logic Device Family Datasheet”, v. 4.3, Feb 2002.
- [7] C.J. Jones, S.J.E. Wilton, “Content-addressable Memory with Cascaded Match, Read and Write Logic in a Programmable Logic Device”, U.S. Patent 6,622,204. Issued Sept. 16, 2003