Content Addressable Memory (CAM) in ATM Applications

XILINX®

XAPP202 (v1.2) January 6, 2001

Author: Marc Defossez

Summary

Content Addressable Memory (CAM) or associative memory, is a storage device, which can be addressed by its own contents. Each bit of CAM storage includes comparison logic. A data value input to the CAM is simultaneously compared with all the stored data. The match result is the corresponding address. A CAM operates as a data parallel processor. CAMs can be used to design Asynchronous Transfer Mode (ATM) switches. Implementing CAM in ATM applications are specifically described in this application note. As a reference, the application note XAPP201 "An Overview of Multiple CAM Designs in Virtex™ Devices" presents diverse approaches to implement CAM in other designs.
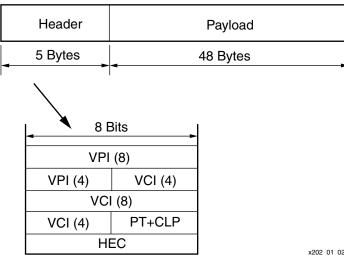
Introduction

A CAM is a memory device used in applications requiring fast searches of a database, list, or pattern. Image or voice systems, computer and communication systems are all users of CAM. CAMs have a performance advantage over other memory search algorithms. This is due to the simultaneous comparison of the desired information against the entire list of prestored entries. CAMs are an outgrowth of RAM technology.

XAPP201 has an overview of CAM blocks versus RAM blocks. It also compares three approaches to designing CAM in Virtex devices. This application note focuses on a large CAM approach for ATM designs.

CAM in ATM

ATM switches, due to their connection based protocol, must translate each ATM cell address at every point along the routing path. As shown in Figure 1, each ATM cell address is contained in two fields in a 5-byte header. The Virtual Path Identifier (VPI) is eight to 12 bits wide. Usually described as a 12-bit word. The Virtual Circuit Identifier (VCI) is 16 bits wide.

Header	Payload
5 Bytes	48 Bytes

8 Bits

VPI (8)	
VPI (4)	VCI (4)
VCI (8)	
VCI (4)	PT+CLP
HEC	

x202_01_022500

Figure 1: **ATM Cell Address**

An addressable space of 26.8 billion possible values is defined by 28 bits of VPI/VCI. Decoding all these values in a short time is impossible. Fortunately, only a few thousand values are active at once in a switch. The switch can maintain a table in memory of the outbound VPI, outbound VCI, and port values corresponding to incoming cells. Figure 2 illustrates this point.

For example, if four thousand connections need to be active in a switch, the VPI/VCI values can be stored in a 4096-word by 28-bit CAM. The resulting 12-bit address output can be used as an index for the RAM table where the translated VPI/VCIs are stored.

	VPI	VCI	Output Port
	12-bit	16-bit	X-bit, most 20-bit or 24-bit
			4K entries

x202_02_073099

Figure 2: **Outbound VPI, Outbound VCI and Output Port Table**

CAM in Virtex Devices for ATM

The Virtex and Virtex-II architectures have two types of RAM internally: distributed RAM and block RAM. Virtex devices can also access external RAM at the common speed of that RAM using Virtex SelectI/O features. This CAM implementation uses the distributed RAM (built from Look-Up Tables or LUTs) for the VPI/VCI table and the block RAM or an external RAM component as the RAM data table.

CAM Bit Table

In this design example the VPI uses 12 bits and the VCI uses 16 bits hence a 28-bit wide search table is essential. The connections list for a CAM is at least 4096 words long. Therefore, the RAM size required is 28×4096 words. There are two types of memory in the Virtex and Virtex-II architectures, distributed RAM and block RAM. Distributed RAM is built using the four, 4-input LUTs in a CLB. It can be configured as a 1-bit by 16 word RAM. Block RAM is an

embedded RAM block that can be configured as a 1-bit by 4096 word, a 2-bit by 2048 word, a 4-bit by 1024 word, an 8-bit by 512 word, or a 16-bit by 256 word RAM.

The suggested approach for this design is to build the CAM in distributed RAM. The implementation of a 1-bit by 4096 words RAM will require 256 RAM. Since this application needs 28 bits by 4096 words, there is a requirement for 7,168 distributed RAM. The XCV400 device can be used for this purpose. The 20 block RAM available in the XCV400 can be used as the output table.

By doing the compare table in distributed RAM, the other logic in the CLB (carry chain, muxes, flip-flops, etc.) is still available to the designer. The Virtex datasheets are a good source of information on the Virtex CLBs. When building the complete CAM described in this application note, the LUTs required are $1.6 \times 7168 = 11,469$ LUTs.

The compare table needs to be initialized with data. This can be done by:

• Initializing at configuration by using the INIT parameter of the distributed RAM.

• Writing to a continuous list of data (block RAM).

• Or a combination of these methods.

No matter how the list is built, using the INIT parameter is always possible. Building the RAM table as a consecutive list may lengthen the search as the list grows. It is prohibitive for even a 4096 word table. The distributed RAM approach appears more useful in this example. The table in Figure 3 is built as a continuous list for initialization and as small parts of 16 entries for the Compare-and-Match operation.
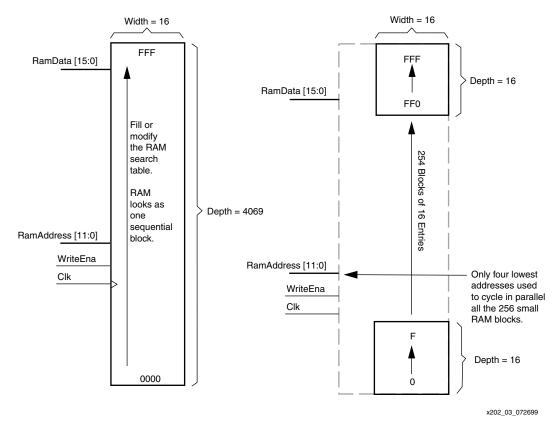

x202_03_072699

Figure 3: **Distributed RAM Table**

CAM Compare (ByteEngine)

The ByteEngine is the basic block of the CAM. The data width size is not a concern when using the LUT approach. A straightforward data width of 16 bits is used in this basic building block (12-bit VPI or 16-bit VCI). A combined VPI/VCI data width is possible (28-bit VPI/VCI).

The ByteEngine in Figure 4 is a small CAM used as often as needed to form the requested CAM size. It has a compare table with 16 entries, the XNOR-gate to do the compare and all the necessary logic for generating a valid and stable Match (HIT) signal.

The RAM table can be initialized using the RamData bus and cycling through the RAM by using the RamAddress bus. Once the table is initialized, the RamWe signal is set to false, and cycle-read can be done through the table. When a compare value drives the DataToComp bus, the XNOR and the Wide-AND gate performs the Compare-and-Match operation. Only when all XNOR are valid is a Match signal produced (Figure 4). A registered and combinatorial output of the match signal is produced. Note: The combinatorial output can produce spikes. Therefore, if it is not registered when the Match signal is produced, it must be registered at a higher level in the design. A simulation of the output of this basic block is shown in Figure 5.
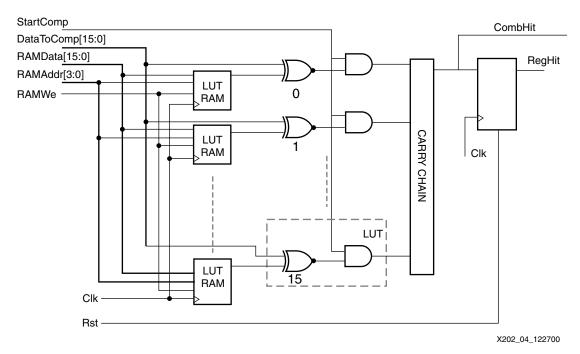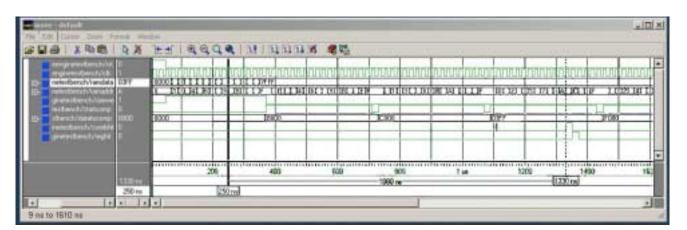

X202_04_122700

Figure 4: **ByteEngine**

Figure 5: **Functional Simulation of ByteEngine**

CAM Size

By using the basic ByteEngine block, any size CAM can be built. Figure 6 details the 256-entry table called EntriesEngine256.
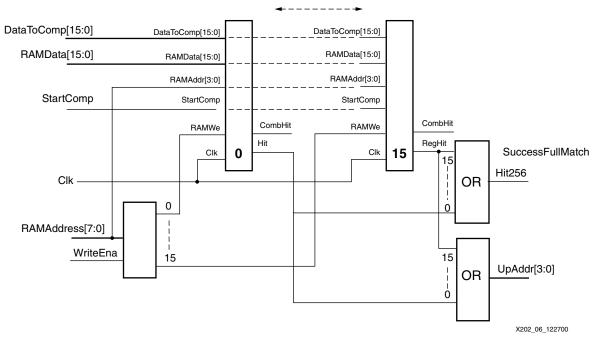

X202_06_122700

Figure 6: **EntriesEngine256 CAM**

EntriesEngine256 is a larger building block for designing large CAMs. It has only the necessary logic to bank select a 16 ByteEngine block at the input, and to generate the Output Address and the Match. The 16 ByteEngine blocks are put together to form a list with 256 entries. An address decoder makes it possible to address (initialize) the list as one long table. On the output, an encoder (Wide-OR-gate) is made for generating the Match signal. A second encoder is made for generating the address where the Match occurred. With Virtex and Virtex-E devices OR-gates are designed using the carry logic. For Virtex-II devices the OR-gates can be made out of normal logic instead of only carry logic. The following example applications use the basic block EntriesEngine256.

XILINX®

CAM example

Figure 7 shows a 16 × 256 CAM using up to 257 Virtex slices. It will run at around 70 MHz in a Virtex device.

Data
DataCompareEn

DataRAMEn

Address[7:0]

RAMOrCnt

DataToComp[15:0]
RAMData[15:0]
RAMAddress[7:0]
WriteEna
StartComp
Clk
Rst

EntriesEngine256

Hit256   MatchSuccess

Upaddr[3:0]

Addr[7:4]   MatchAddress[7:0]
Addr[3:0]

CycleCnt[3:0]

StartCompare

CycleCnt

Clk

Rst

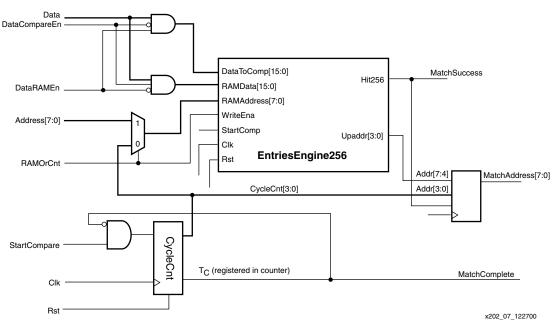T_C (registered in counter)   MatchComplete

x202_07_122700

Figure 7: **16 × 256 CAM**

Figure 8 shows a cycle diagram. The assumptions for the CAM are as follows:

- If DataToCompare = "1" and DataToRun = "0", then data can be clocked into the DataToCompare register.
- If DataToRAM = "1" (the DataToCompare register is disabled) then data is passed to the RAM.
- If RAMOrCnt = "0", then the CycleCnt is passed to the RAM for reading the contents.
- If RAMOrCnt = "1", then the addresses are passed to the RAM.
- To fill the RAM table: DataToRAM and RAMOrCnt must both = "1".
- Make DataToCompare = "1" and DataToRAM = "0" and set RAMOrCnt = "0".
- The "value_to_compare_to" can be loaded into the register.
- Make DataToCompare = "0" afterwards
- Start the cycle counter by bringing the signal StartCompare = "1".

When this is done, the cycle counter (CycleCnt) reads (CycleCnt) through the RAM data and compares it against the DataToComp. When a match is found in one of the 16 banks (for 256 deep), the match register of that bank is set. The value is decoded and a Hit256 signal is generated. Hit256 stops the cycle counter. MatchSuccess loads the "address composer" logic. The address that matches the incoming data is generated out of the decoding of the bank where the Hit is found and the state of the counter. Hit256 and StartCompare enable an AddressMatch register to load the valid address.

When no Hit is found and the Cycle counter reaches the end, a CompleteMatch cycle signal is generated and the Cycle counter is stopped.
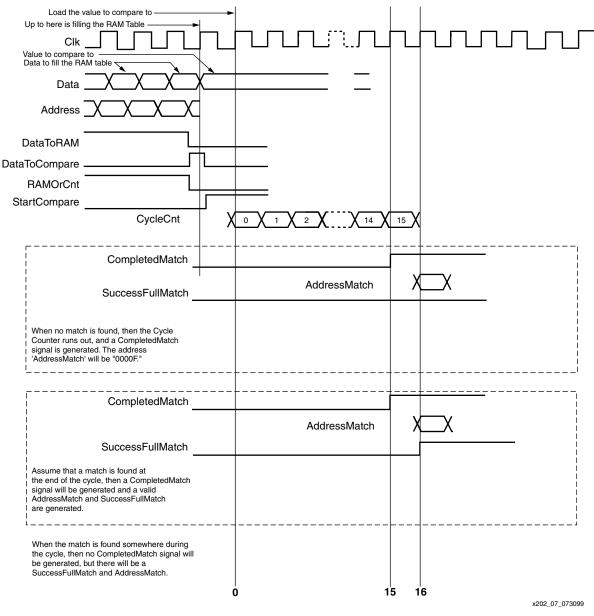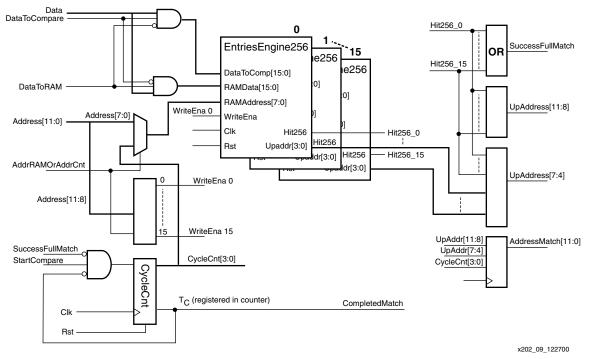
Figure 8: **Cycle Diagram**

Figure 9 describes a 4096 word CAM built in the same manner using 16 basic EntriesEngine256 modules and more decoding logic.

Figure 9: **4096 Word CAM**

Figure 9 outlines a complete CAM solution using the block RAM as the output data table. The data in the output table can be easily modified when using the full Dual Read/Write Port™ capabilities of the block RAM.
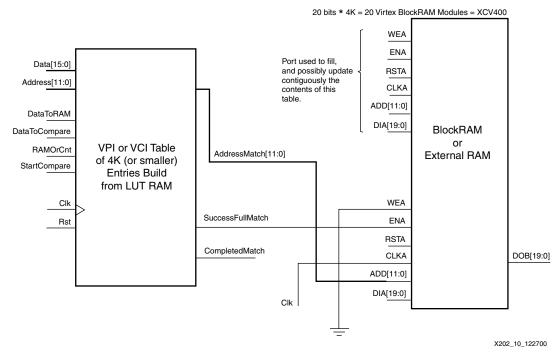

Figure 10: **Complete CAM Solution**

Summary

CAM Design

- Use the 1×16 distributed RAM in an arrangement of 4096 words (or less). Look-up the contents of the RAM in 16 clock cycles (Synchronous RAM).

- When a match is found, the generated address is used to select data in the block RAM or in the external RAM outside the FPGA.

- Between two Compare-and-Match operations, the Search RAM Table can be easily adapted. Since it appears as a normal consecutive RAM and only the address and data are needed to write to a specific location in the table.

- By using the full Dual Read/Write Port capabilities of the block RAM, the data stored in the block RAM can be independent of the modified search table. For ATM the data stored in the block RAM is the Output Port.

- This design is not concerned about the size of the data or the size of the Search Table because there is a new data look-up in every 16 clock cycles. Some extra cycles are needed to latch the VPI or VCI address and to output the address found in the register (maximum of 18 cycles).

- A CAM with 4096 entries will fit into a XCV600 or a XCV600E. This will use all 24 block RAMs available as a 24-bit by 4096 word data table.

- Smaller CAMs such as an 256 words by 80 bits can be made entirely with distributed RAM.

- Both the data to be compared and compare tables need to be initialized before using the CAM. When initialization is done during operation the following occurs:

  - The compare table in the distributed RAM needs to be switched to the continue RAM configuration mode. During normal CAM operation, this memory is divided into smaller words (16 words).

  - The compare table in the block RAM can be updated at any time by using the second port. The block RAM is a true Dual Read/Write Port RAM with two completely separate ports.

Conclusion

This CAM design enables a look-up every 18 cycles. Sixteen of these cycles are needed to scroll through the small distributed RAM blocks. One clock cycle is needed to load the data to be compared and one cycle is needed to output the matched value.

As demonstrated by the application note XAPP201, the flexibility of the Virtex and Virtex-II series is a key advantage when designing a CAM. In addition to the solution in this application note, XAPP203 and XAPP204 offer different approaches based upon different application needs.

The most economical way of making a large CAM in ATM applications is to use both the distributed RAM (basic configuration 1×16) and the block RAM (basic configuration 1×4096) available in both Virtex and Virtex-II architectures. The distributed RAM and an external RAM block can also be used to make a large CAM. The CAM compare table can be made using the distributed RAM while the data can be stored in the block RAM or an external RAM. A CAM with a 24-bit by 4096 word compare table will fit into the XCV600 or the XCV600E.

Appendix A: Synthesizable HDL Code Reference Design

This appendix describes a hierarchical, synthesizable design implementing a search engine or CAM in Virtex slices.

The complete HDL code, the simulation files for ByteEngine, EntriesEngine and the 16×256 CAM are available in the reference design (File: **xapp202.zip** or **xapp202.tar.Z**).

The header of each VHDL module is listed below:

Module: MatchMachine4k.vhdl

```
-- Entity Name: MatchMachine4k
-- File Name:    MatchMachine4k.vhd
-- File Path:    D:\projects\Cam\vhdl\
-- Project :
--
-- Purpose:  This is a machine that can do a CAM operation
--           on 16 bits for 4096 entries in 18 clock cycles.
--           files used :
--                   ByteEngine.vhd
--                   EntriesEngine256.vhd
--
---- Authors: Marc Defossez
--
-- Tools: Synplicity 5.2.1
--
-- Revision History:      Created:        20/04/99
--                        Last opened:    Wednesday, 06 June 99
--
--
--
--   Disclaimer:  THESE DESIGNS ARE PROVIDED "AS IS" WITH NO WARRANTY
--                WHATSOEVER AND XILINX SPECIFICALLY DISCLAIMS ANY
--                IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR
--                A PARTICULAR PURPOSE, OR AGAINST INFRINGEMENT.
--
--   Copyright (c) 1999 Xilinx, Inc.  All rights reserved.
----------------------------------------------------------------------
```

Module: MatchMachine256.vhdl

```
-- Entity Name:  MatchMachine256
-- File Name:    MatchMachine256.vhd
-- File Path:    D:\projects\Cam\vhdl\
-- Project :
--
-- Purpose:  This is a machine that can do a CAM operation
```

```
--                on 16 bits for 256 entries in 18 clock cycles.
--                files used :
--
--                         ByteEngine.vhd
--                         EntriesEngine256.vhd
--
...
--------------------------------------------------------------------------
```

Module: EntriesEngine256.vhdl

```
-- Entity Name:   EntriesEngine256
-- File Name:     EntriesEngine256.vhd
-- File Path:     D:\projects\Cam\vhdl\
-- Project :
--
-- Purpose:      This is the engine that compares in 16 clock
--               cycles 256 values against a given value on a
--               double byte width (16 bits).
--               This is one section of a VPI/VCI cam.
--               Makes use of
--                         Byte Engine.vhd
--                         EntireEngine256.ucf
--
...
--------------------------------------------------------------------------
```

Module: ByteEngine.vhdl

```
-- Entity Name:   ByteEngine
-- File Name:     ByteEngine.vhd
-- File Path:     D:\projects\Cam\vhdl\
-- Project :      CAM
--
-- Purpose:     Engine over 16 bits.
--              Compares 16 bits over 16 deeh and give a Hit
--              signal if the 16 bit value is found in to table.
--
--              Because the depth will be bigger than 16 bit's there
--              is need for working in BANKS of 16.
--              Like for 256 entries, 16 banks will be needed.
--              In the file above this, two banks are combined.
--              Reason for doing this is RLOCing.
--
--              As the ByteEngine is made now, 8 CLBs are in this way:
--              If nicely lined up, there will be a column of 8 CLBs where
```

```
--         slice S1 is used to store 2 x a RAM16X1S (16 bits).
--         and slice S0 will only contain 8 LUTs + carry chain for the
--         comparitor. Thus there is some mismatch between the RAM
--         column hight and the comparitor hight.
--
--         For UCF file test purposes, following is done
--         Combination of two of these ByteEngine.vhd files is done
--         in TwoBanks.vhd and a UCF file with RLOC's is made
--         (TwoBanks.ucf)
--         A small 256 entries engine is made, lateron this 256 engine
--         can be combined to form bigger chuncks of memory.
--
--
...


End of Appendix A.
```

Revision History

The following table shows the revision history for this document.

Date	Version	Revision
09/01/99	1.0	Initial release.
09/23/99	1.1	Initial Virtex_E Updates
01/06/01	1.2	Updated for Virtex-II series, added Figure 5, changed Figures 4, 6, 7, and 9.