Reconfigurable Content-Addressable Memory

Zoltan Baruch
Computer Science Department
Technical University of Cluj-Napoca
26-28 Gh. Baritiu St., 400027 Cluj-Napoca
Romania
Zoltan.Baruch@cs.utcluj.ro

Cristina Savin
Computer Science Department
Technical University of Cluj-Napoca
26-28 Gh. Baritiu St., 400027 Cluj-Napoca
Romania
savin@cs-gw.utcluj.ro

Abstract – **Content-Addressable Memories (CAMs) are parallel search circuits, typically found in embedded circuitry. This paper presents a reconfigurable implementation of a CAM on an FPGA (Field Programmable Gate Array) device using the JBits package. This package provides software control for all configurable FPGA resources. The result obtained is a fully-interrogatable CAM, with support for run-time reconfiguration. This approach produces a fast and very flexible alternative to traditional implementations. Timings obtained are similar to those of embedded hardware circuits, which makes the core faster than previous reconfigurable CAM cores. A simple application was designed for testing the core functionality, which has proven its utility in search-intensive applications.**

I. INTRODUCTION

Content-addressable memories represent hardware search engines used in search-intensive applications. They offer significant improvements compared to other search methods (for example, binary or tree-based search) due to the fact that they enable parallel comparison of the search pattern to all entries in the stored list. Such memories are identified by the content of data themselves, rather than an address.

Due to limitations concerning costs, CAM use has been restricted to certain niche applications, embedded on custom designs, especially processor caches. In the last few years, many new opportunities of using CAM properties have been identified, including Ethernet address lookup, data compression, database accelerators, pattern recognition, neural networks, high-bandwidth address filtering, routing, security, or information encryption on a packet-by-packet basis for high-performance data switches, firewalls, bridges, and routers [1], [2].

In this paper we present the design and implementation of a reconfigurable CAM on an FPGA device. This implementation makes use of the run-time reconfiguration capabilities offered by the JBits package. JBits offers an application programming interface to Xilinx FPGA bitstreams, allowing the design and dynamic reconfiguration of Xilinx Virtex devices [3]. Due to the run-time reconfiguration support, logic and routing resources can be dynamically tailored to the application requirements [4].

An implementation of a CAM with the JBits package is described in [5]. This approach uses the reconfiguration capabilities of JBits for replacing flip-flops with LUTs. Data is written to the CAM by reconfiguring the LUT contents. A different approach is considered in this paper. The CAM cell includes a flip-flop and additional circuitry for the matching. The structure was designed as to reduce as much as possible the required connection wires, since the routing resources are limited. The matching functionality is more complex than that of [5], supporting a fully-interrogatable configuration. The price paid for flexibility is that of increased complexity.

The organization of this paper is as follows. Section II presents a background for content-addressable memories, as well as an analysis of the JBits package and its associated tools. Section III describes the implementation of the fully-interrogatable CAM. The testing procedure used for validating the functionality of the CAM is presented in Section IV. A comparison between our implementation and a previous one is provided in Section V. Finally, the conclusions related to our implementation and possible future developments are presented in Section VI.

II. BACKGROUND

A. Content-Addressable Memories

As opposed to random-access memories (RAMs), in which the stored data are identified by means of a unique address assigned to each data word, content-addressable memory words are identified by their content. CAMs are very useful in applications where intensive search operations are to be performed.

In general, a search for a particular piece of data in a RAM with n words will take a time of $t * f(n)$, where t is the time taken to fetch and compare one word of the memory and f is an increasing function of n. Hence, with an increase in n, the search time increases too. In the case of a CAM with n words, the search time is almost independent of n because all the words may be searched in parallel. Only one cycle time is required to determine if the desired word is in memory, and, if present, one more cycle time is required to retrieve it [6]. To be able to perform a parallel search, each data word needs to have a dedicated circuit, which increases significantly the cost of CAMs.

For some implementations, it suffices to know whether or not the searched word is found in the memory. Other implementations also supply the location where the data word was found.

Based on the function used for determining a match, content-addressable memories can be classified into *exact match* and *comparison* CAMs. An exact match CAM identifies data based on equality with certain key data. In a comparison CAM, the search is based on a general comparison, using various relational operators. Usually, different parts of the memory word can be used as the key. In the most general case, any part of the word can be used for the matching.

Based on the values that can be stored, there are two types of CAMs: *binary* and *ternary* [7]. Binary CAMs can only store binary digits ('0', '1'), while ternary CAMs can store binary digits as well as "don't care" values ('X').

Ternary CAMs may have a global mask as well, which allows the search pattern to also contain "don't care" values. This is useful when the width of the search pattern is small, so that two or more entries can be stored in the same CAM location.

CAM words have two parts. The most important part is the *search field*, which is the part of the word that is matched with the search pattern. Another part is the *return field*, which is the information returned during a read operation. This field may contain related information or an index.

Fig. 1 shows the block diagram of an m-word, n-bits per word CAM [6]. Before the search process is started, the word to be searched is loaded into the argument register A. The segment of interest from the word is specified by setting the corresponding bits of the key/mask register K to '1'. Once the argument and key/mask registers are set, each word in memory is compared in parallel with the content of the argument register. If a word matches the argument, the corresponding bit of the match register M is set. Since any set of the n bits of the argument can be selected as key, it is possible that several memory words yield a match. In this case, the selection circuitry ensures that only one word is chosen, based on a certain priority policy.

Two priority policies can be used [8]:

- *Inherent priority.* In this case, the CAM words are stored in order of priority. For example, by using a priority encoder, the top address of the CAM may have the highest priority and the bottom address may have the lowest priority.
- *Explicit priority.* An explicit priority field is added to each CAM word. In case of a multiple match, the word with the highest explicit priority, as stored in the priority field, is returned.

The advantage of explicit priority is that updating the CAM becomes easier, since a new word can always be added after the last word in the CAM. When using inherent priority, an address has to be reserved by shifting down other words and updating the memory that is addressed by the CAM [8].

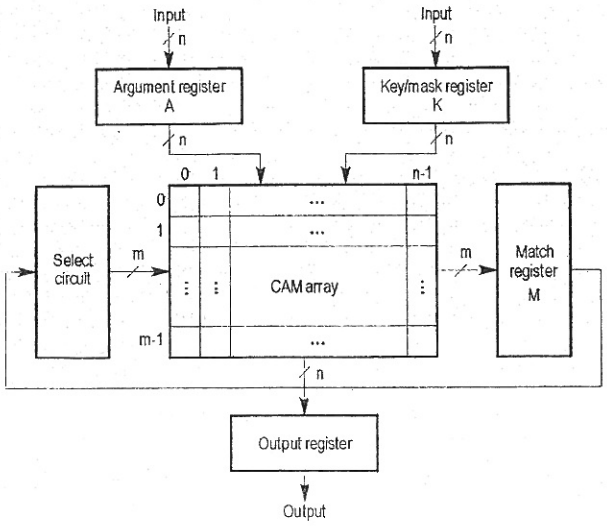

Fig. 1. Block diagram of a content-addressable memory

B. The JBits Package

The JBits package consists of a set of Java classes that provide an API to the configuration bitstream of the Xilinx Virtex FPGA devices. It allows software control for configuring the device's resources, such as CLBs, BRAMs, IOBs, and routing elements. This package can operate on bitstreams generated by CAD tools or on those read back from the device itself.

JBits may be used as a stand-alone tool or as a base to produce other tools, including traditional place-and-route CAD applications, as well as more application-specific tools. The resulting model is actually a Java code, which makes the software portable and it allows a better integration within the main system. As an example, a graphical user interface (GUI) can be constructed for the reconfiguration application in a simple way, maintaining the consistency between the software interface and the FPGA device.

The JBits package is organized on several layers. The Bit-Level Interface provides access to all the configurable resources of a Virtex device, including the look-up tables (LUTs) inside each Configurable Logic Block (CLB) and the routing resources adjacent to the CLBs. This interface is the lowermost layer that allows to set or clear a single bit or a group of bits in the configuration bitstream of the device. The device architecture is represented as a two-dimensional array of CLBs, and each CLB is referenced by a row and column.

The Bit-Level Interface interacts with the *Bitstream* class, which manages the device' configuration bitstream and provides support for reading and writing bitstreams from and to files. This class can also read back the existing configuration data from the operating device, which is necessary for dynamic reconfiguration [4].

There are four main functions provided by the Bit-Level Interface. The *read()* and *write()* functions allow configuration bitstreams to be read or written. The *get()* function allows to query the state of a programmable logic resource, while the *set()* function allows to set the state of a programmable logic resource to a specified value. The Bit-Level Interface also contains a series of constants which define each programmable resource of the device and the values they can be set to.

Another component of the JBits package is the *Run-Time Parameterizable Core* (RTPCore) library. It contains a set of classes that define macrocells or cores representing the most common circuit elements, such as counters, adders, registers, and other standard Xilinx Unified Library logic and computation functions. These cores can be dynamically parameterized and relocated within the device. Currently, JBits provides Unified Library primitives for the Virtex CLB and Block RAM resources. In addition to these primitive cores, other non-primitive RTP cores can be used, which are created by instantiating primitive or non-primitive subcores connected with nets and busses [9].

The *Java Run-Time Reconfiguration* (JRTR) API allows small changes to be made directly to the Virtex device, without interrupting operations. Such changes can be done much faster than with usual methods. The JRTR API keeps track of the changes done in the configuration and only the necessary data is rewritten to the device. A much more detailed description of specific aspects of the Virtex devices related to reconfiguration are presented in [2].

The *Xilinx Hardware Interface* (XHWIF) offers a portable layer for connecting the JBits API to the FPGA-based hardware. This interface standardizes the way that JBits applications communicate with the hardware, so that using the same interface, applications can communicate with a variety of boards. All the hardware specific information is hidden inside of a class that implements the XHWIF interface.

XHWIF contains methods for describing the type and number of FPGA devices on the board, for configuring the devices, for reading back the configuration memory of the devices, for incrementing the on-board clock, and for reading and writing from/to on-board memories, if they are available.

The XHWIF interface is implemented as a server application. This server allows other applications to communicate with reconfigurable computing boards located anywhere across the Internet. This capability allows multiple users to access a board and to debug designs using tools such as *BoardScope*, without having direct access to the hardware.

BoardScope is an interactive debug tool for Virtex devices. It provides a simple and powerful interface for visualizing the contents of FPGA circuits, during their operational state [10]. It features a CLB-based design view mode, which displays the output state of all CLBs and allows interactive probing of internal CLB state. The waveform display mode allows to view signals and busses in a way similar to that used by circuit simulators. *BoardScope* uses the XHWIF interface to communicate with the FPGA-based hardware.

The *Virtex Device Simulator* (*VirtexDS*) is also part of the JBits package and provides a software model of the entire Virtex family of FPGA devices. The approach taken is to simulate at the device level, by providing an interface much alike the actual hardware [11]. By operating at the device level, *VirtexDS* simulates the actual FPGA device, and therefore provides a high level of simulation accuracy. It also allows to identify illegal configurations that would damage the actual hardware, which is important for debugging run-time reconfigurable applications. The device simulator interface is identical to that of the actual hardware, which allows existing applications, including the *BoardScope* debug tool, to interface directly to the simulator with no modifications.

III. DESIGN AND IMPLEMENTATION

All developed cores are included in the CAM package. This includes several sub-packages. The most important one, which gathers the cores used for the actual CAM implementation, is *CAMCores*. The other two packages define the *TestCores* and the test classes used for validating the functionality of cores in the first one.

The implementation uses the classical model for the CAM cell, with some additions included for the purpose of reducing as much as possible the required wires. In this way, cells can be cascaded both horizontally through the match lines, and vertically through the *Q* lines. A block diagram of the CAM cell implemented in a CLB is shown in Fig. 2. Cascading the cores adds some overhead to the operations, since the signals must propagate throughout the CAM.

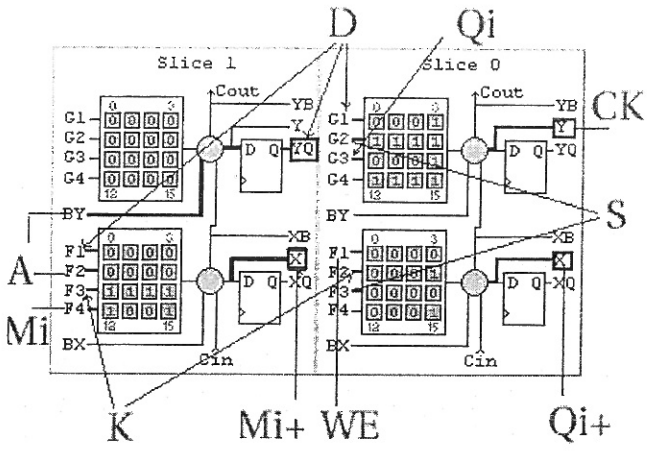

Fig. 2. The CAM cell implemented in a CLB

In the JBits implementation, the LUT4 core from the *ULPrimitives* package was used for generating the Mi+, Qi+ outputs and the clock for the D flip-flop. Consequently, the cell requires three LUTs and one flip-flop. A different CLB was used for implementing each cell. Thus, the cell has CLB granularity and the core width and height are 1.

Fig. 3 illustrates the structure of a cell inside the CAM array. Some special cases concern the first row and the leftmost cells. A simple way of dealing with these special cells would be to connect Qi or Mi to the default values ('0' and '1', respectively). This variant was used at the beginning of the development process, for simplicity reasons. Some problems could be identified for this approach. The most important is that it required some external cores for generating the constant values, extending the device space needed for the implementation. This is why a different approach was preferred, namely defining some special cores for dealing with these cases. As a result, the number of input pins is reduced for these cells, by eliminating Qi and Mi, respectively.

CAM cells are grouped as words in the CAM core. The cells are lined horizontally, thus the height of the core is 1, while the width is given by the word size, specified by the width of the input busses *A* and *K*. The CAM array is defined in a manner similar to that used for the CAM word, by grouping together CAM words vertically. This configuration was preferred since it achieves a uniform management of these cores and eases the connections to the other components.

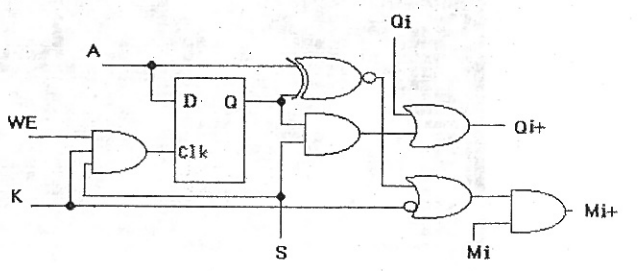

Fig. 3. Structure of the CAM cell

A predefined selection circuit, namely the *OneHot* core, exists in the

com.xilinx.JBits.Virtex.RTPCore.CAM

package. Initially, this core was used as selection circuit, but conflicts occurred related to some CLB resources. The cause of this problem seems to be related to the automatic placement of cores with different granularities. This behavior was not documented in the JBits API, but the tests performed during development indicate that the *addChild* method of the *RTPCore* class places elements of smaller granularity over the neighboring component. This problem was solved by extending the dimension of the neighboring core with 1.

Although an equivalent result would have been obtained by modifying the *OneHot* width accordingly, building an entirely different core was preferred. The new selection circuit is called *Selection*, and it is also included in the *CAMCores* package. It is based on the selection cell defined by the *SelCell* core.

The *CAMBigBox* core connects everything together. CAM words are placed one in top of the other, in the left side of the core. The selection circuit and the match register are placed to the right of the CAM array. An output register is added for buffering the output bus *Q*.

IV. TESTING PROCEDURE

All cores in the *CAMCores* package were tested in a similar fashion. For each such core, a *TestCore* class was defined. This is a core which instantiates the component to be tested and also provides the means to supply the values for the input busses and nets. For this purpose, the *TestInputVector* class from the package

com.xilinx.JBits.Virtex.RTPCore.TestGeneration

was used. This core can be used for providing inputs to the test cores. In one version, the values for the core outputs are written to the BRAM from a specified external file, provided as parameter when implementing the core. The final version of the CAM core defines several input files containing the test vectors.

The classes from the *CAMTesters* package are used for generating the actual bit files and the associated .cft files, which are then imported into *BoardScope* for debugging and testing. Some simple testbench applications were also designed, for testing partial reconfiguration behavior.

The *CellTestCore* in the *TestCores* package instantiates a *CAMCell*, a *TestInputVector* to provide the required inputs, and a *Register* as output buffer. A similar structure is used for testing the CAM word, the CAM array, and the selection circuitry.

In addition to the *TestCore* and its associated *Test* class, for the final test of the CAM core we used a testbench application, illustrating the values in the match register and the contents of the output buffer, for a succession of values of the argument *A* and the key *K*.

The core requires two cycles per operation: the first for selecting a certain word from the memory (the first word to match the masked *A* vector), and the second for the actual operation – read or write. Some problems occurred when trying to test the *CAMBigBox* core. The reason is related to the implementation of the selection circuit. The *Selection* core is built by cascading several selection cells, thus different delays are introduced for the selection lines of the CAM array. This causes some glitches and activates some additional words. This problem was solved by delaying the *WE* signal until the falling edge of the second cycle. In this way, writing is performed only after the selection lines are stable, and the results are those expected.

Fig. 4 shows the state view of the core in the *BoardScope* window.

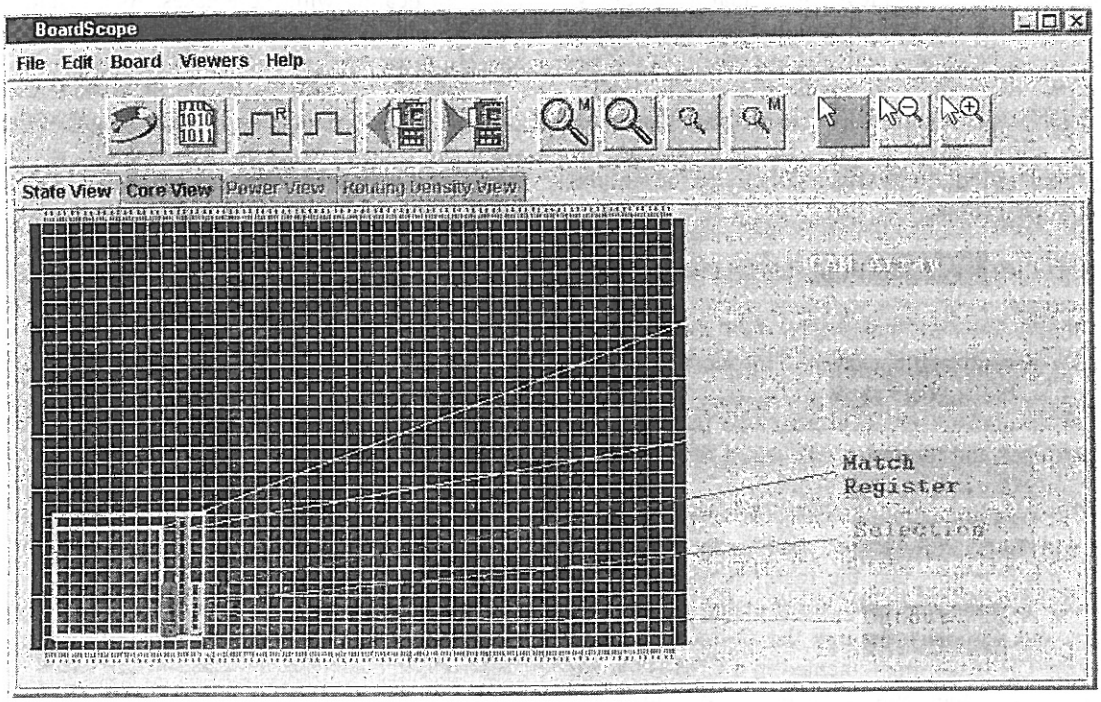

Fig. 4. The CAM core in the BoardScope window

V. EXPERIMENTAL RESULTS

We have compared our implementation with that presented in [5]. The main difference between the two versions of CAM consists in the way JBits run-time reconfiguration (RTR) support is perceived and used. The implementation of [5] uses RTR as an essential feature, while our implementation only uses RTR to dynamically change the CAM parameters (memory size and word width).

This approach was taken for flexibility reasons. The CAM described in [5] can only be used to search for a value, yielding a simple found/not found answer, while our implementation allows to search for any pattern of the same length as that of the CAM word. The contents of the memory word are also obtained as output. On the other hand, the CAM described in [5] can store patterns (words including the "don't care" value), by defining an appropriate LUT function, while our CAM only stores binary values. This is one of the causes why our implementation requires more resources on the device.

The major advantage of our approach is related to the way write operations are performed. Contents of the CAM described in [5] can be modified only by reconfiguring the LUTs, which requires a significant time. In contrast, our implementation allows the write operation to be performed strictly at the hardware level. Therefore, the speed of the circuit is similar to that of common hardware implementations, with two cycles per operation.

The supplemental features provided by our CAM implementation justify the additional resources required, making it possible not only to design classic lookup-type components, but also to use it in more complex applications, such as data compression, pattern-recognition, neural networks, or high-bandwidth address filtering.

Since currently the JBits API does not provide support for timing analysis, the timing differences between the two implementations cannot be measured.

VI. CONCLUSIONS

In this paper we have described a run-time reconfigurable and fully-interrogatable CAM. The run-time reconfiguration feature allows the core to be tailored to the particular hardware design and to dynamically adapt to changes within the system. In addition, the JBits package allows to embed the core in more complex FPGA designs.

Compared to other existing implementations, the core-based design using the JBits package provides much more flexibility. Although a different design of a reconfigurable CAM is described in [5], our design considers an alternative approach, which is closer to the structure within CAM chips. The strengths of this core reside in its fully-interrogatable nature and in the way the read and write operations are performed. Disadvantages are mainly related to the increased complexity of the CAM cell. All in all, the reconfigurable CAM presented provides the features required by fast search-intensive applications such as signal tracking and processing [12], routing, data compression, or real-time artificial intelligence.

Future work concerns the development of an application providing an interactive interface to the core. Another direction of study concerns the integration of the developed core in a more complex networking application. Some additional research could also be done to find the possibilities of reducing propagation delays when cascading several cells, of either the selection circuit or the memory array, by using specific architectural features of the target device.

VII. REFERENCES

[1] S. Azgomi, "Using Content-Addressable Memory for Networking Applications", http://www.commsdesign.com/main/1999/11/9911feat3.htm, accessed May, 16, 2004.

[2] S. McMillan and S. A. Guccione, "Partial Run-Time Reconfiguration Using JRTR", in *Proceedings of the 10th International Workshop on Field-Programmable Logic and Applications*, FPL 2000, pp. 352-360.

[3] S. A. Guccione and D. Levi, "A Java-Based Interface to FPGA Hardware", in *Configurable Computing: Technology and Applications*, 1998, pp. 97-102.

[4] S. A. Guccione, D. Levi and P. Sundararajan, "JBits: A Java-Based Interface for Reconfigurable Computing", in *Second Annual Military and Aerospace Applications of Programmable Devices and Technologies Conference*, MAPLD, 1999.

[5] S. A. Guccione, D. Levi and D. Downs, "A Reconfigurable Content Addressable Memory", in *Parallel and Distributed Processing. Proceedings of the 15th International Parallel and Distributed Processing Workshop*, IPDPS 2000, pp. 882-889.

[6] Z. F. Baruch, *Structure of Computer Systems*, U.T. PRES, Cluj-Napoca, Romania, 2002, pp. 174-185.

[7] K. Pagiamtzis, "Content-Addressable Memory Introduction", http://www.eecg.toronto.edu/~pagiamt/cam/camintro.html, accessed March, 2, 2004.

[8] J. M. Ditmar, "A Dynamically Reconfigurable FPGA-based Content Addressable Memory for IP Characterization", Master's Thesis, KTH - Royal Institute of Technology, Stockholm, Sweden, 2000.

[9] Xilinx Inc., "JBits Tutorial", JBits SDK Version 2.8, 2001.

[10] S. A. Guccione and D. Levi, "BoardScope: A Debug Tool for Reconfigurable Systems", in *Configurable Computing: Technology and Applications, Proceedings of SPIE 3526*, 1998, pp. 239-246.

[11] S. McMillan, B. J. Blodget and S. A. Guccione, "VirtexDS: A Virtex Device Simulator", in *Reconfigurable Technology: FPGAs for Computing and Applications II, Proceedings of SPIE 4212*, 2000, pp. 181-187.

[12] A. Annovi, M. G. Bagliesi, A. Bardi, R. Carosi, M. Dell'Orso, P. Giannetti, G. Iannaccone, F. Morsani, M. Pietri, and G. Varotto, "A Pipeline of Associative Memory Boards for Track Finding", in *IEEE Transactions on Nuclear Science*, vol. 48, no. 3, June 2001, pp. 595-600.