TU Clausthal

Clausthal University of Technology

Efficiently Reprogramming Boolean Functions by sTCAM/RAM

Harald Richter and Dietmar Sommerfeld

IfI Technical Report Series **IfI-06-07**

Department of Informatics

Clausthal University of Technology

Impressum

Publisher: Institut für Informatik, Technische Universität Clausthal
Julius-Albert Str. 4, 38678 Clausthal-Zellerfeld, Germany
Editor of the series: Jürgen Dix
Technical editor: Wojciech Jamroga
Contact: wjamroga@in.tu-clausthal.de
URL: http://www.in.tu-clausthal.de/forschung/technical-reports/
ISSN: 1860-8477

The IfI Review Board

Efficiently Reprogramming Boolean Functions by sTCAM/RAM

H. Richter, D. Sommerfeld

Institute of Computer Science,
Clausthal University of Technology, Germany
richter@in.tu-clausthal.de

Abstract

A device for boolean mappings $\mathbf{B}^n$ -> $\mathbf{B}^m$, $\mathbf{B}$={0,1}, n, m ∈ N is given that is hardware efficient and quickly reprogrammable, combining the best features of look-up tables and programmable logic arrays (PLAs). It is especially suited to implement functions with large n, m. The device's setup is given on the transistor level. Its hardware complexity is compared to look-up tables and PLAs by means of a detailed cost model.

1 Introduction

Boolean functions f_b: $\boldsymbol{B}^n$ -> $\boldsymbol{B}^m$, $\boldsymbol{B}$={0,1}, $n, m \in N$ are the base for all logic circuitry including digital computers. In this paper, the realization of boolean functions is improved with respect to reprogrammability and required transistor count by a new hardware. Both improvements are of interest in practical applications such as reconfigurable computing [7], [9], dynamic finite state machines and cellular arrays. The current state of the art is such that many types of programmable logic devices (PLDs) are commercially available and capable of implementing f_b with $n, m \leq 32$. PLD types are 1.) programmable array logic (PAL), 2.) programmable logic array (PLA), and 3.) field programmable gate arrays (FPGAs). In this paper, we restrict to the PLA type. Although algorithms and software exist to effciently map f_b onto PLAs [5],[6], fast reprogramming during runtime is still an issue, besides the amount of transistors (or gates) required for a PLA.

An efficient reprogrammable PLA is obtained e.g. by using a (Flash)RAM as a look-up table for f_b. This approach is efficient in several cases because RAMs can be built with high transistor density, low cost and fast read/write access. In read mode, programmed RAM content is addressed by using a (sub)set of input vectors $\underline{in} \in \boldsymbol{B}^n$ as RAM address

bits. Each corresponding output vector $\underline{out} \in \boldsymbol{B}^m$ is stored in one RAM cell, with m, n as the number of input/output bits. In write mode, (re)programming takes place before or during operation which means that the look-up table is dynamically updated. However, the key drawback of the RAM approach is its exponential transistor complexity of $O(m2^n)$. Thus, large n (n>32) are not realizable by a look-up table.

On the contrary, PLAs are efficient in hardware resources by mapping a minimized representation of f_b (mostly the sum of AND terms) onto two arrays of AND/OR gates forming a two-tiered logic on the chip. However, PLAs are slower than RAMs with respect to programming. For instance, programming 64 KBytes in a RAM takes 640 µs, if its write cycle time is 10 ns and n =16 and m = 8 is assumed. Programming a 16 input and 8 output PLA may take an order of magnitude longer. This is because write access to the PLA's internal configuration RAM is not as fast as writing explicitly to a standard RAM. Additionally, the algorithmic process to condense f_b to minimal form is NP-hard from a mathematical point of view and neither unambiguous nor one-to-one [3], [8]. Dynamic reprogramming during run time is excluded by the compute time required. Therefore, implementing f_b in a quickly reprogrammable and hardware thrifty way is still an open issue. In this paper, a two-tiered solution is proposed that uses a specially designed version of a ternary content addressable memory on its input stage and a standard SRAM of a size smaller than in look-up table approach on its output stage.

In section 2 of this paper, the proposed hardware and mapping algorithm for f_b is explained. In section 3, a transistor-count cost-model is given to compare RAM and PLA to the proposal. Section 4 summarizes the results.

2 Proposed Device

Before the proposed device is presented its theoretical background must be explained. This is accomplished by giving a formal specification of f_b. There are three methods to specify f_b : 1.) By explicitly enumerating every input vector $\underline{in}$ and its corresponding output vector $\underline{out}$ in a truth table. Each table entry contains one distinct bit combination of n bits, together with its corresponding m output bits. For a complete specification, the table may contain up to 2^n entries with $n+m$ colums each. 2.) By a textual notation defining all states in a way such as: IF <input condition x> THEN <issue output 1> ELSE <issue output 2>. State notations are used by software tools like ABEL [1], e.g., and 3.) By boolean logic with AND/OR terms.

The truth table method as well as boolean notation is used here. For our purpose, the truth table must be augmented by three-valued logic and an OTHERWISE statement to simplify table entries. Compact tables are mandatory because the target applications of the proposed device are identified by large n, m, for example n = 64 and m = 32. We use the sum-of-products method (SOP) for boolean notation in which f_b is defined as a vec-

tor of subfunctions f_i such that $f_b = (f_1, f_2,\ldots,f_m)$. Each f_i sets or clears one of the m output bits of f_b. This defines f_b indirectly by m output bits *out(i)* at position i, $i \in \{1,2,..., m\}$. f_i is also called output term. *out(i)* is defined by a SOP as:

$$out(i) = f_i = z_{i,1} + z_{i,2}\ldots + z_{i,k(i)} \text{ for } 1 \le i \le m \text{ and } 0 \le k(i) \le 2^n,$$
and "+" as the *OR*-operator

Please note that index k may have the special value 0 wich means that *out(i)* is a constant. The $z_{i,k}$ component of the subfunction f_i is called product term. The parameter *k(i)* denotes how many product terms are necessary to define *out(i)*. At most, such many product terms are needed for *out(i)* as binary combinations over n bits exist, i.e. 2^n. A product term is computed by

$$z_{i,k} = y_{i,1} * y_{i,2} * \ldots * y_{i,n} \text{ and } * \text{ as } AND\text{-Operator}$$

The $y_{i,j}$ component of the $z_{i,k}$ product term is called input term. An input term defines how an input bit *in(j)* at position $j \in \{1,2,..., n\}$ has to be treated. There are three possibilities for $y_{i,j}$: 1.) Take the input bit *in(j)* as it is, 2.) Invert the input bit, or 3.) Disregard the input bit, i.e mask it out by a „don't care". Thus, $y_{i,j}$ is $\in \{in(j), \overline{in}(j), -\}$, representing three-valued logic.

Operation Principle of the Device

The proposed hardware implementation of f_b that is quickly reprogrammable and hardware thrifty is not based on minimizing the number *k(i)* of product terms, but on data compression. The key idea is to identify input vectors *in*, to code these vectors and to compress their code. This means that the set of vectors that is input to f_b over the course of time is considered as a 1-D array of bits upon which lossless data-compression is applied. This new view is one answer to the principal question what entropy a boolean function contains. The compression algorithm used in this paper is code book compression. Code book compression identifies recurrent patterns in a time sequence of bits. Each pattern that occurs at least two times is entered into a code book, which may be implemented as a 2D table. The pattern is associated to an unique table index. For data compression, a repetitive pattern is replaced by its index every time when the pattern occurs. Compression is effective when the index is shorter than the pattern itself. For example: 8 different patterns of a length of 16 bits may occur two or more times in a bit string. Then, the index for each pattern requires 3 code bits, which is less than 16 bits. For decompression, the index is replaced again by the code book entry that contains the 3 bit code and the 16 bit equivalent. Please note, that the condition for the table index restricts the usability of code book compression to a subset of applications. However, this subset may comprehend many practical cases.

Code book compression means for the truth table of f_b that every input vector is considered as potentially repetitive and can be applied to f_b several times. The code book of f_b is a table of entries with constant length n. Because of the three-valued nature of the in-

put terms $y_{i,j}$ each code book entry is $\in$ {*in(j), $\overline{in}$(j),* -}.

Both, the recognition of repetitive patterns and the code book itself can be implemented by a specially designed content-addressable memory that manages ternary input data. In the following, this device is called sTCAM (special ternary content-addressable memory). Ternary logic is handled by sTCAM such that for every bit of a search key a mask bit exists that may switch the search key bit to the don't care state. If the mask bit is set the search key bit is set to don't care. One difference between usual TCAMs [2] and sTCAM is that the latter does not store any data except the search keys itself, and consequently never outputs data. Its only output is the address of the location where a search key presented to its input is found. If no hit occurs the sTCAM issues a reserved address to indicate the search miss. A search key that is input to the sTCAM is compared simultaneously with all preprogrammed content in *e* words. sTCAM can be considered as an inverse RAM to which content is input and address is output. The sTCAM has 2*n* input bits multiplexed on *n* lines, *e* internal memory words of 2*n* bits for search keys and don't cares, and *c* output bits to indicate which address location contains the input in case of a hit. Subsequently, the *c* output bits are used as input for a standard RAM that stores all output vectors of f_b. The resulting hardware setup is shown in figure 1.

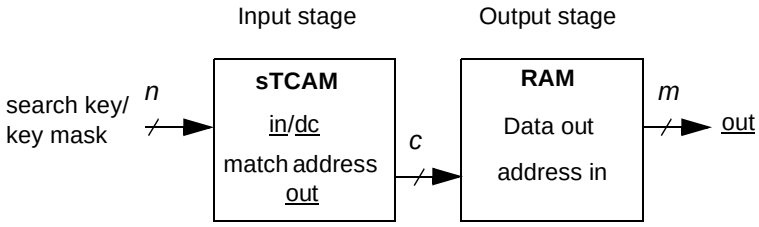

Fig. 1: Reprogrammable device implementing any boolean function f_b: $\boldsymbol{B}^n$ -> $\boldsymbol{B}^m$.

Internal Setup of sTCAM

For sTCAM, SRAM cells are used as memory instead of DRAM for reasons of speed and the guarantee of real-time operation. SRAMs require no write back phase after read and need no periodic refresh during which normal operation would be suspended. A complete sTCAM memory cell comprises two SRAM cells, one for the stored search key (ssk) and one for the don't care state (dc), an EXOR bit comparator, and a standard CMOS NOR gate. The cell setup is depicted in figure 2. In part a) of this figure, a SRAM bit cell is depicted, comprising two cross-coupled CMOS inverters and two enable transistors for I/O. In part b) the used NOR gate is shown, and in c) the complete TCAM bit cell is given. The ssk bit is coupled to the bit comparator in a different way

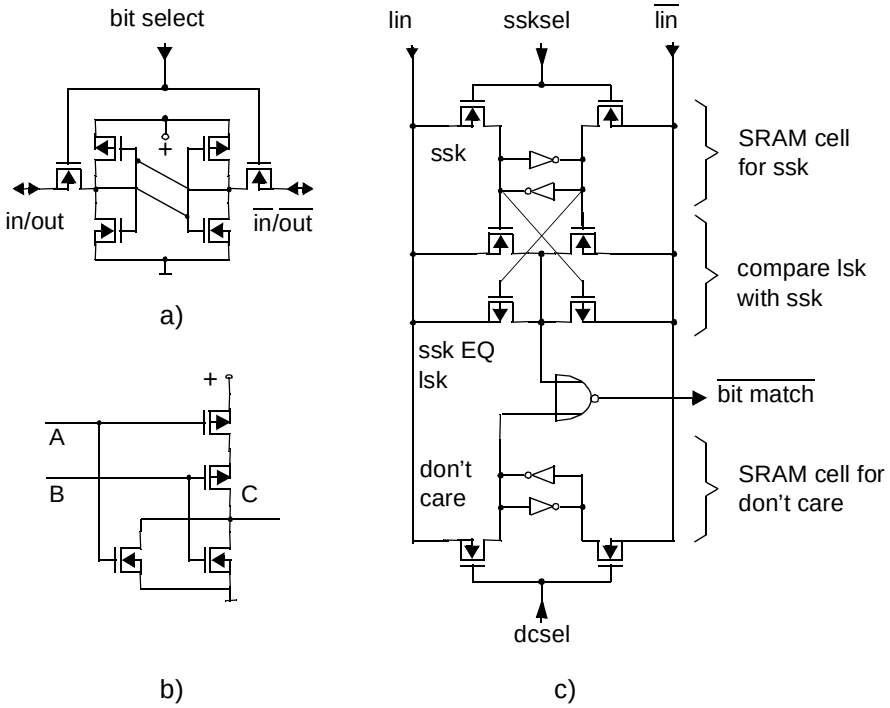

Fig. 2: a) SRAM bit cell, b) NOR gate, c) Setup of a sTCAM bit cell.

as in standard SRAM because sTCAM memory is never read out.

The function of sTCAM is as follows: the ssk bit is compared with the latched search key bit (lsk) that is presented to the cell via the in and $\overline{\text{in}}$ lines. If both match or if the don't care bit is set, a bit match indication is output ($\overline{\text{bit match}}$). To initialize a bit cell, the don't care and stored search key select signals (dcsel/ssksel) have to be activated sequentially. To program the ssk bit, the ssksel signal has to be activated, while the dcsel is deactivated and the in/$\overline{\text{in}}$ lines have to be set appropriately. For programming the don't care bit, dcsel must be activated and ssksel deactivated, and the cell's content must be present at the in/$\overline{\text{in}}$ lines. The in and $\overline{\text{in}}$ lines are tristated during read operation indicating a normal state and sTCAM's only output is $\overline{\text{bit match}}$. Both is different compared to a SRAM.

Each word in a sTCAM chip comprises *n* bit cells (bc) connected together as indicated in figure 3. The individual bit match signals of a word are ANDed together by a wired AND. Thus, a word match signal is obtained. Wired ANDs and wired ORs are used throughout the whole design to save transistors. The dcsel/ssksel signals of a bit cell are connected with their counterparts of the same word so that all word bits are selected and

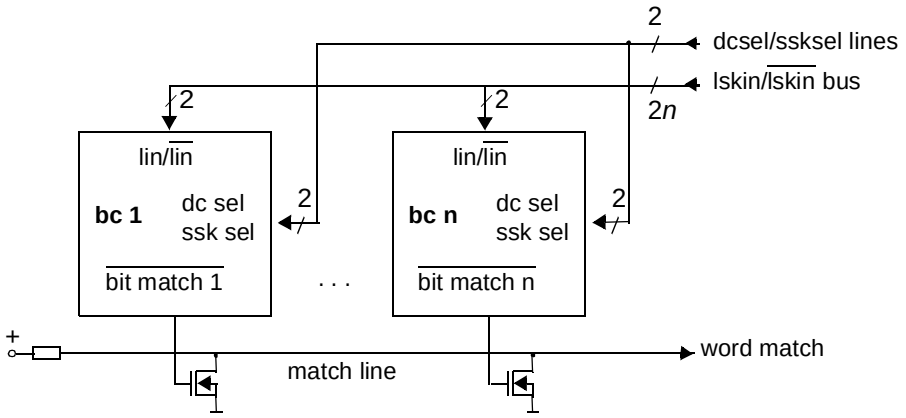

Fig. 3: Setup of a sTCAM word of *n* bits.

written simultaneously by activating 2 lines only. However, every pair of lskin/$\overline{\text{lskin}}$ of the bits in all words are connected together to a single bus of 2*n* lines spanning across the whole chip. This allows search keys of *n* bits to be input via the chip's *n* in/$\overline{\text{in}}$ signals. The dcsel/ssksel lines of each word are kept separately, since words must be programmed individually with data supplied via the common lskin/$\overline{\text{lskin}}$ bus.

In total, a sTCAM chip has *e* words of *n* bits each ($e,n \in \mathbb{N}$). Please note, if these words were arranged physically one after the other, a huge address decoder of 2*e* outputs for write select would be necessary. Additionally, a large encoder of 2*e* inputs for word match detection and address outputting would be required too. Furthermore, for bit cell placement on the chip a rectangular area is mandatory for chip design reasons. Therefore, the *e* words are arranged in a 2D array of *r* rows and *s* colums with $e = r \cdot s$, ($r,s \in \mathbb{N}$). This significantly saves transistors in the decoder and encoder. The *s* match signals of the words of a row are ORed together to a single row match signal. Equivalently, the *r* match signals of a column are ORed to a single column match signal. Now, the detection whether a word has matched the search key requires two match address encoders with $r+s$ outputs instead of one with rs outputs. The benefit is because of the fact that $r+s << rs$ for larger *r*, *s*. See figure 4 for the 2D arrangement of row and column match address encoding. The same argument holds for addressing the memory array in the write case, i.e. for programming a bit cell. In figure 5, the 2D arrangement for write selection is given.

The overall setup of the proposed sTCAM differs from a normal SRAM in several aspects: sTCAM consists of 2 input registers to hold search key and key mask, *r* rows and *s* columns of bit cells of 20 transistors each, plus 4 more transistors for address decoding and match encoding, and two address encoders for the match word address, and two decoders to program the chip. The block diagram of sTCAM is shown in figure 6. Rows

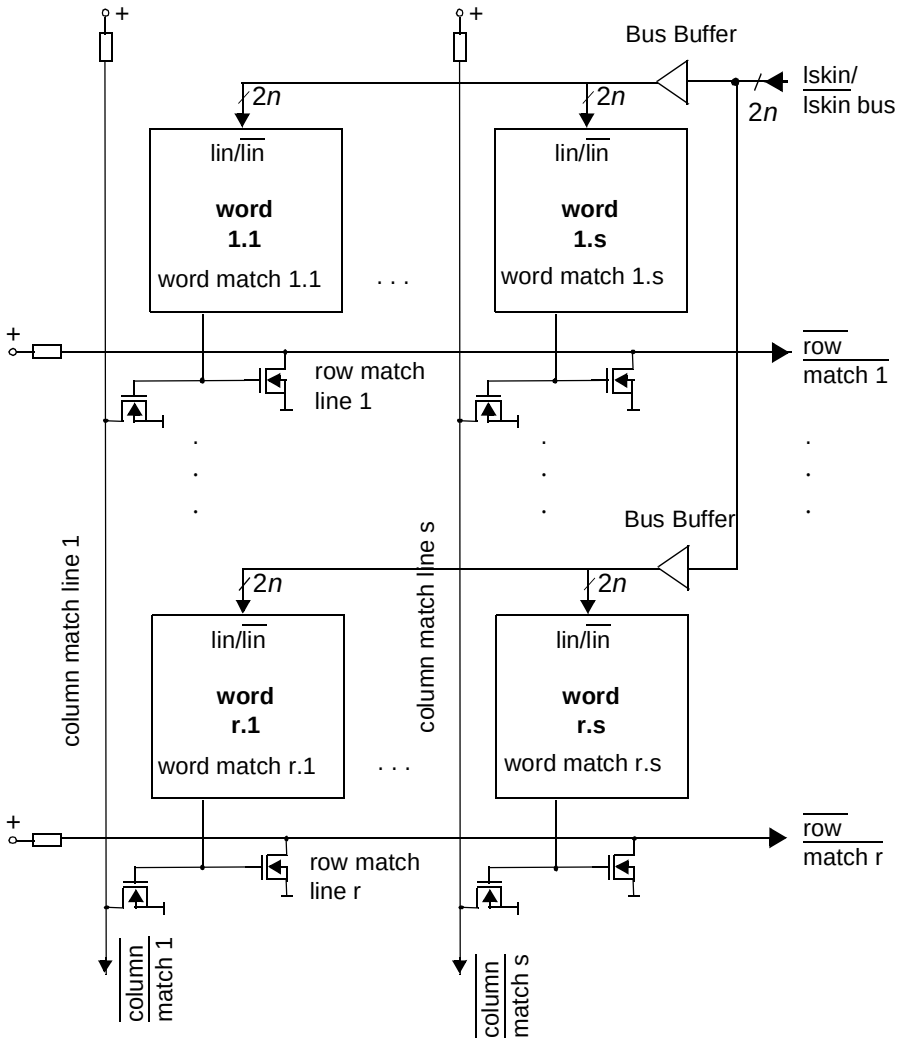

Fig. 4: 2D array of sTCAM words for row and column match address encoding.

are addressed by $r' = \lceil \log_2 r \rceil$ bits, columns by $s' = \lceil \log_2 2s \rceil$ bits. The column match encoder needs $s'' = \lceil \log_2 (s+1) \rceil$ input bits to distinguish between matched column address or sTCAM miss. Decoders and encoders are critical components with respect to hardware complexity. For e = 4 M for instance, an array of 2048x2048 words has to be addressed requiring one row decoder of 2048 AND gates with 11 inputs and one column decoder of 4096 AND gates and 12 inputs each, together with 11+12 inverters. The

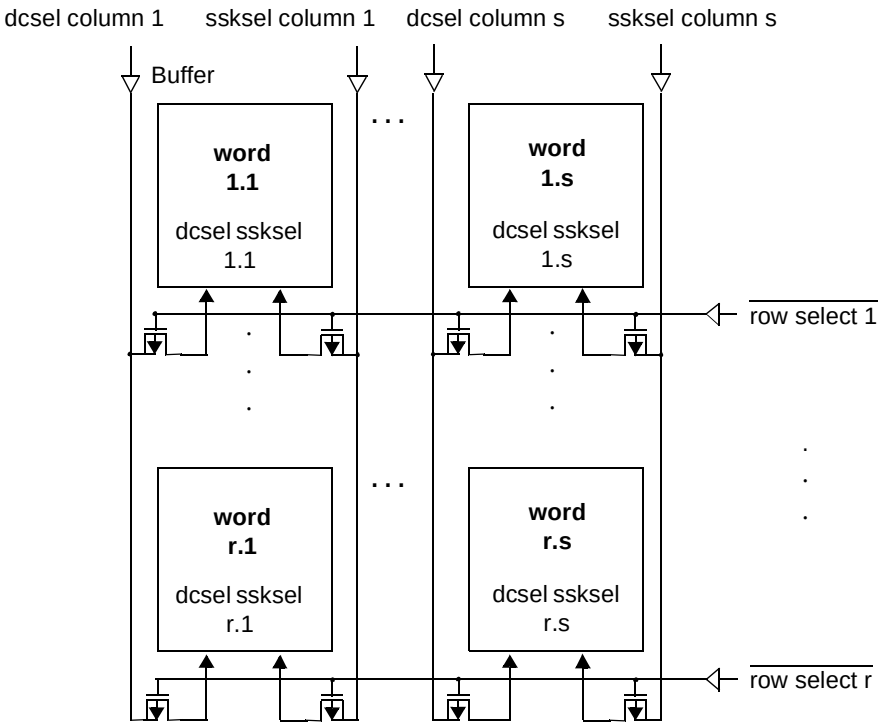

Fig. 5: 2D array for row and column write decoding.

AND gates are accomplished economically by wired ANDs consisting of 1 transistor per gate and input and 2048 or 4096 resistors. For a general array of size *rs*, we obtain for the row decoder $r\log_2 r+2\log_2 r+r$ transistors. (In CMOS chip technology, a resistor is implemented as a transistor with its gate connected to a fixed potential, and an inverter is implemented by 2 transistors). The hardware amount above reduces to $r(1+\log_2 r)$ if the inverters are neglected. Similarily, $2s(1+\log_2 2s)$ transistors are required for the column decoder.

For the encoders, we propose a structure explained by means of the example of 8 -> 3 encoding (figure 7): At each cross (x), a transistor is located as part of a wired OR. The bottom row forms a wired OR of 8 inputs which outputs H level if no sTCAM match has occured. The row above the bottom row detects whether the most significand bit (MSB) of the matched address is cleared or set. It is set only, if no L is present in the lower half of all match signals. The MSB-1 output is H if no signal in the first or third quarter of all columns is L. The least significant bit (LSB) finally, is set if no signal with even address (0, 2, 4, 6) is L. The transistor count for the row encoder is (1/

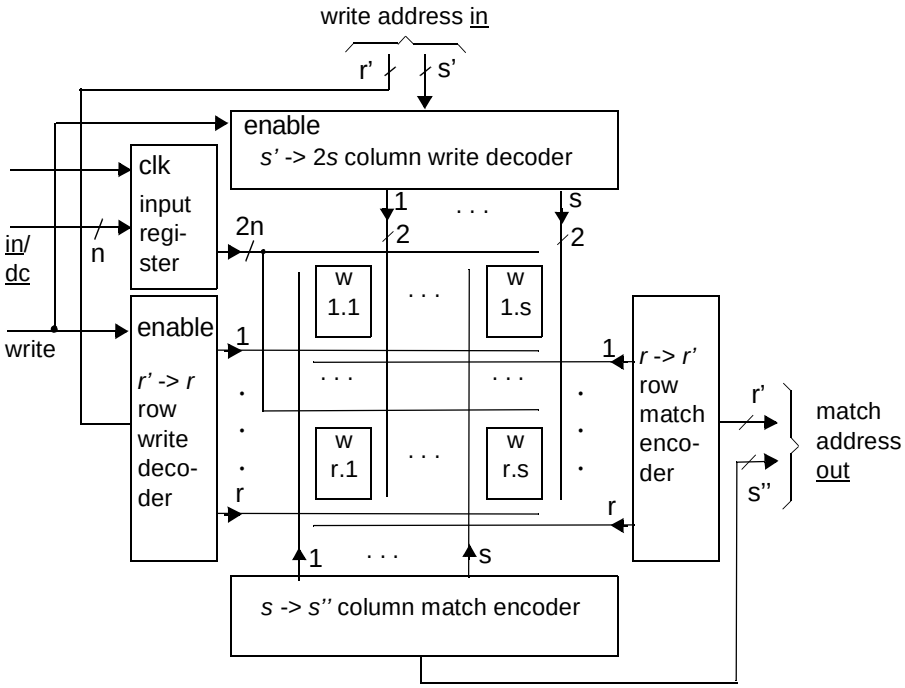

Fig. 6: Block diagram of a sTCAM chip.

2)$r\log_2 r+\log_2 r+1+r$ which reduces to $r(1+0.5\log_2 r)$ for large r. Similarily, we have $s(1+0.5\log_2 s)$ for the column encoder. Encoders and decoders together require $r(2+1.5\log_2 r)+2.5s(2+\log_2 s)$ transistors

Before sTCAM can be used, all required e search keys and key masks have to be input to sTCAM while setting the write line and writing sTCAM line addresses. Additionally, all 2^c, $c= r'+s''$ output vectors of m bits each specified by the user have to be programmed into the RAM. For normal operation, the R/$\overline{W}$ signal is set to high, the write signal is set to low, and no write addresses are required. In figure 8, the sTCAM programming is illustrated in contrast to RAM programming.

Preparing the Truth Table of f_b

The truth table of f_b must be prepared before sTCAM and RAM can be programmed as explained. The process is illustrated by means of an example (Table 1).

1.) Define f_b by means of a table comprising $\leq 2^n$ rows and an OTHERWISE statement.

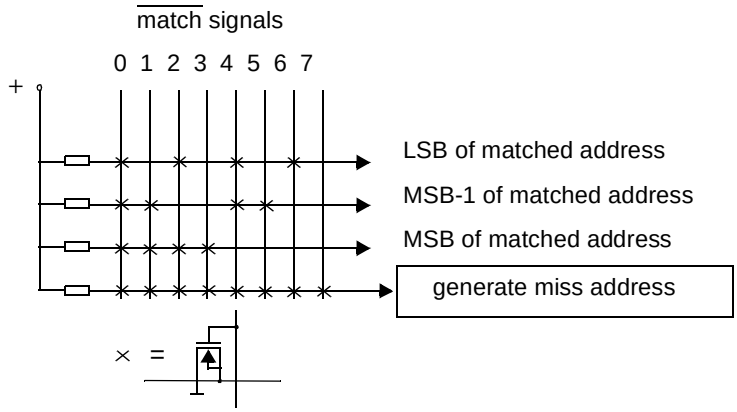

Fig. 7: Example of an efficient implementation of an address encoder.

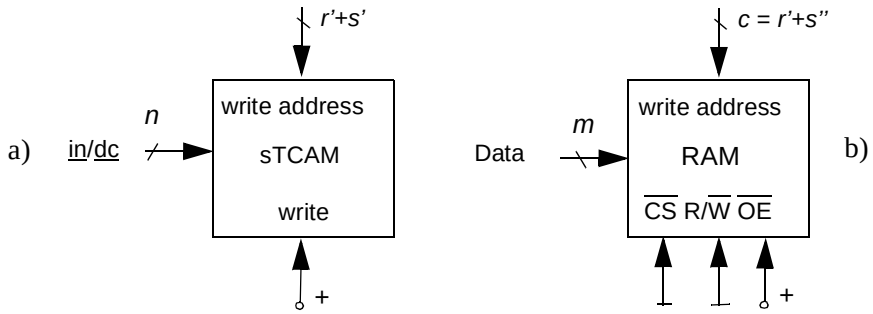

Fig. 8: Programming sTCAM (a) in comparison to RAM (b).

2.) Regard only rows with at least one output bit set and delete those rows where the output vector contains only zeroes, provided the remaining rows comprise ≥ 50% of all previous rows. Otherwise, regard only rows with at least one output bit cleared and delete those rows where the output vector contains only ones, and treat the output bits as active low signals (negative logic).

3.) Replace any two entries with identical output vectors that have input vectors of Hamming distance H=1 (=difference by 1 bit) by a single entry containing a „-“ at the bit position where the two entries differed.

4.) Repeat step 3.) until no more identical output vectors exist with input vectors of H=1. Then, a simplified truth table results with a minimum number of entries required to code f_b (Table 2).

Input			Output		
0	0	0	0	0	1
0	0	1	0	1	0
0	1	0	0	1	1
0	1	1	0	1	1
OTHERWISE			0	0	0

Table 1: Example of f_b: $\boldsymbol{B}^3$ -> $\boldsymbol{B}^3$ given as a truth table with OTHERWISE statement.

Input			Output		
0	0	0	0	0	1
0	0	1	0	1	0
0	1	-	0	1	1

Table 2: Simplified truth table comprising only 3 rows.

5.) If the number $r \in \boldsymbol{N}$ of the remaining input vectors is $\geq 2^{n-1}$ then stop, because the code book index together with the missing entry address would require the same number of code bits as the input vectors, i.e. n bits. No data compression is possible. Else continue.

6.) Cast the input vectors of the simplified truth table into a sTCAM by programming one input vector after the other into subsequent sTCAM words. The special sTCAM output 111...1 is reserved for a search key miss.

7.) Cast the output vectors into a RAM consisting of $r+1$ words with m bits each by programming successively one output vector after the other into subsequent RAM cells. Additionally, programm a zero vector as RAM output for the missing entry address. (This turns into a vector of ones if negative logic is used.)

After the seven steps, the RAM outputs the appropriate vector for each code book index, i.e. for each input vector, as well as a zero vector (vector of ones for negative logic) for all other inputs (Table 3).

sTCAM Input			sTCAM Output		RAM Output		
0	0	0	0	0	0	0	1
0	0	1	0	1	0	1	0
0	1	-	1	0	0	1	1
OTHERWISE			1	1	0	0	0

Table 3: Casting of f_b: $\boldsymbol{B}^3$ -> $\boldsymbol{B}^3$ into sTCAM/RAM.

3 Cost Model

In the following, we model the costs of sTCAM, look-up table and PLA by counting their transistors on the chip. Let the user's truth table specify e ($0 \le e \le 2^n$) boolean mappings $\boldsymbol{B}^n$ -> $\boldsymbol{B}^m$. A PLA that implements these mappings needs n inputs and m outputs and t different AND terms ($t \le e$) of n bits that are ORed together to deliver one output bit. This bit behaves as prescribed in the truth table for all e entries. Without restriction, no two output bits that are identical in their behavior are allowed. This is because the ORing together selects m different subsets out of the set of all subsets of AND terms. Since the set of all subsets has 2^e elements $m \le 2^e$ holds. In the PLA's AND array, a horizontal wire is allocated for every different AND term, together with a vertical wire for every input. In the PLA's OR array, we have t horizontal and m vertical wires. At each crossing of a horizontal with a vertical wire, a transistor may be positioned, thus forming a wired AND in the AND array and a wired OR in OR array for that horizontal line. The AND array consists at most of en transistors, as well as of e resistors and n inverters. The OR array has em transistors, m resistors and no inverters. The total transistor count for the PLA is

$$T_{PLA} \le e(m+n+1)+2n+m$$

yielding an $O((n+m)e)$ hardware complexity.

A SRAM that implements e boolean mappings needs n address inputs and m data outputs and 2^n different memory cells to be able to output data for every of the 2^n input combinations. A commercial example of an SRAM is given in [4]. SRAMs consist of a word array of 2^n words of m bits organized in r rows and s columns with $r = s = 2^{n/2}$ if n is even (quadratic array), or $r = 2^{(n+1)/2}$ and $s = 2^{(n-1)/2}$ if n is odd (slight rectangular array). In the following, let us assume without imposing any restriction that n is even.

Such a SRAM has two $n/2$ -> $2^{n/2}$ decoders, $m2^n$ bit cells of 6 transistors each, m sense/ write circuits of 4 transistors each, m data register bits of 6 transistors each, $2m$ input/ output buffers and a few logic gates. For the row and column decoders of the RAM, we require approximately $2^{n/2}(2+n)$ transistors. For the memory array, $6m2^n$ transistors are required. All other components are neglectable for $n>5$ which is the case for all products commercially avaliable. Therefore, the transistor count for SRAM is

$$T_{SRAM} \approx 6m2^n+2^{n/2}(2+n)$$

which is an $O(m2^n+n2^{n/2})$ hardware complexity.

A sTCAM that implements e mappings $\boldsymbol{B}^n$ -> $\boldsymbol{B}^m$ needs n inputs and $\log_2 e$ data outputs and e different memory cells. The differences between SRAM and sTCAM in terms of transistor count are the following: 1.) The memory array of size rs does not correlate with the number of input bits n, instead an arbitrary number of e words of n bits is possible. 2.) The bit cell has 24 transistors instead of 6. 3.) The column decoder is of double size. 4.) There are two additional match encoders. Furthermore, for reasons of real estate size on the chip a (nearly) rectangluar memory array is also appropriate as in the case of RAM. Therefore, $r^2 = s^2 = e$ can be assumed without restriction. This sums up to

$$T_{sTCAM} \approx e(24n+2\log_2 e).$$

The RAM for the sTCAM has $\log_2(e+1) \approx \log_2 e$ inputs and m outputs which gives

$$T_{sTCAMRAM} \approx 6me+0.5e(2+\log_2 e).$$

The total transistor count for sTCAM and RAM is thus

$$T_{total} \approx e(24n+6m+3\log_2 e),$$

which is an $O(e(n+m+\log_2 e))$ hardware complexity.

4 Conclusion

The hardware complexity is by far the highest for the look up table approach and the least for the PLA. However, the proposed sTCAM/RAM's complexity is only sligthly worse than that of a PLA. It differs in higher proportional factors of 24 and 6 compared to 1 and an additional $\log_2 e$ term. This term grows slowly for large e. The sTCAM/ RAM has the advantage against PLA that it is much quicker reprogrammable than a possible PLA's configuration RAM because it is directly RAM based. Additionally, sTCAM/RAM allows larger m since SRAMs are available to word lengths of up to 64 bits. On the other hand, it allows for larger n compared to the look-up table since no exponential complexity is present. This leads to the conclusion that for all cases where fast reprogrammability is required the sTCAM/RAM solution is better than the look-up table and the PLA, especially for large e,n,m.

References

[1] http://www.seas.upenn.edu/ese/rca/software/abel/abel.primer.html.

[2] Sherrie Azgomi, *Using content-addressable memory for networking applications*, http://www.commsdesign.com/main/1999/11/911feat3.htm, 1999.

[3] Robert K. Brayton et.al.: *Logic Minimization Algorithms for VLSI Synthesis*, Kluwer Academic Publishers, 1984.

[4] Cypress Semiconductor Corporation, *CY7C109D Document # 38-05468*, 2005.

[5] Nicholas Pippenger, *The Shortest Disjunctive Normal Form of a Random Boolean Function*, Random Structures & Algorithms, Volume 22, Issue 2, pp.161-186, John Wiley & Sons, 2003.

[6] George Vastianos, *Boolean Functions' Minimisation Software based on the Quine-McCluskey method*, Software Notes, Athens, 1998.

[7] Harald Richter, Christian Siemers: *Efficient Reprogrammable Architecture for Boolean Functions and Cellular Automata*, 6th International Workshop on Boolean Problems, Freiberg, Germany, 23.-24. Sept. 2004, Editor: B. Steinbach, ISBN 3-86012-233-9, Medienzentrum der TU Freiberg, 2004.

[8] Ingo Wegener, *The Complexity of Boolean Functions*, Wiley-Teubner, 1987.

[9] Christian Wiegand, Christian Siemers, Harald Richter: *Definition of a Configurable Architecture for Implementation of Global Cellular Automaton*, ARCS 2004, LNCS 2981, pp.140-155, Springer Verlag, 2004.