SAKARYA UNIVERSITY OF APPLIED SCIENCES
INSTITUTE OF GRADUATE EDUCATION

DESIGN AND IMPLEMENTATION OF FPGA-BASED ASSEMBLER USING CONTENT ADDRESSABLE MEMORY (CAM)

M.Sc THESIS

Abdelkader LAZZEM

Department	**:**	**ELECTRICAL AND ELECTRONICS ENGINEERING**
Supervisor	**:**	**Prof. Dr. İhsan PEHLİVAN**
Co-Supervisor	**:**	**Assoc. Prof. Dr. Halit ÖZTEKİN**

January 2023

SAKARYA UNIVERSITY OF APPLIED SCIENCES
INSTITUTE OF GRADUATE EDUCATION

DESIGN AND IMPLEMENTATION OF FPGA-BASED ASSEMBLER USING CONTENT ADDRESSABLE MEMORY (CAM)

M.Sc THESIS

Abdelkader LAZZEM

Department : ELECTRICAL AND ELECTRONICS ENGINEERING

This thesis was accepted by the committee given below on 04/01/2023 by unanimously/majority of votes in Institute of Graduate Education, Sakarya University of Applied Sciences.

COMMITTEE	STATUS
Head of Committee: Prof. Dr. İhsan PEHLİVAN	SUCCESSFULL
Member: Asst. Prof. Ali Furkan KAMANLI	SUCCESSFULL
Member: Asst. Prof. Gülüzar ÇİT	SUCCESSFULL

DECLARATION STATEMENT

All the data in the thesis are obtained by me within the framework of academic rules, all visual and written information and results are presented in accordance with academic and ethical rules, there is no manipulation in the data used, in case of benefiting from the works of others, it is referred to in accordance with scientific norms, I declare that it has not been used in any thesis work. I accept all kinds of legal responsibility that may arise in case of a situation contrary to this statement.

Abdelkader LAZZEM

04/01/2023

ACKNOWLEDGEMENTS

I would like to extend my acknowledgment and sincere gratitude to all the people who helped me complete my master's degree.

Firstly, I would like to express my deepest appreciation to my esteemed supervisor, Professor Dr. İhsan PEHLİVAN, for encouraging me and providing me with all the necessary facilities to complete my thesis. His guidance and valuable suggestions helped me a lot during my research.

Secondly, I would also like to extend my special thanks and appreciation to my esteemed co-supervisor, Assoc. Prof. Dr. Halit ÖZTEKİN, whose knowledge and valuable experience I benefited from during my master's degree studies. The completion of my thesis would not have been possible without his support and guidance. He did not hesitate to assist me in obtaining the needed information and supported me throughout the process, from planning to writing the thesis.

Thirdly, I would like to express my gratitude to my thesis committee, Asst. Prof. Ali Furkan KAMANLI and Asst. Prof. Gülüzar ÇİT, for their helpful comments and suggestions during my thesis progress.

Finally, I would like to extend my thanks and appreciation to my family and friends for being my source of strength and support in achieving my academic goals and for always being by my side in all circumstances. Their faith in me kept my spirits high and motivated me throughout my education.

CONTENTS

ABBREVIATIONS

ASCII	: American Standard Code for Information Interchange
ASIC	: Application-specific Integrated Circuit
ASM	: Assembly Language
BiCAM	: Binary Content Addressable Memory
BL	: Bit line
BRAM	: Block Random-access Memory
CAM	: Content Addressable Memory
CISC	: Complex Instruction Set Computer
CLB	: Configurable Logic Block
CPLD	: Complex programmable logic devices
CPU	: Central Processing Unit
DRAM	: Dynamic Random-access Memory
FF	: Flip-flops
FPGA	: Field-programmable Gate Array
GPU	: Graphics Processing Unit
HDL	: Hardware Description Language
IP	: Intellectual Property
LUT	: Look-up Table
MLSA	: Match-line Sense Amplifier.
Opcode	: Operation code
RAM	: Random-access Memory
SRAM	: Static Random-access Memory
TCAM	: Ternary Content Addressable Memory
VGA	: Video Graphics Array
VHDL	: VHSIC Hardware Description Language
VHSIC	: Very High-Speed Integrated Circuit
WL	: Word line

LIST OF TABLES

LIST OF FIGURES

DESIGN AND IMPLEMENTATION OF FPGA-BASED ASSEMBLER USING CONTENT ADDRESSABLE MEMORY(CAM)

SUMMARY

An assembler is a program that converts the assembly language into executable machine code to perform basic processor operations. Where memory plays a crucial role in the conversion process, as it is used to store and retrieve the mnemonic instruction sets required to operate the system. Then, Machine language instructions are extracted from these instructions and delivered to the central processing unit (CPU) or microcontroller to be executed.There are various search algorithms, including linear, binary, and hashing-based algorithms, which are developed to speed up the search operation, which can be a critical factor in increasing the overall efficiency of the system. These algorithms are sharing a point which is they all software-based techniques, not hardware-based ones. In this thesis, we propose a novel approach to enhance search operations by applying a hardware-based modification. The proposed modification involves replacingcommonly used memories like Random-access Memory (RAM) with Binary Content Addressable Memory (BiCAM). BiCAM is a hardware-based parallel memory that can perform the search operation in a single clock cycle. The FPGA-based BZK.SAU assembler which is an assembler designed for educational purposes was chosen as the case study for this thesis, and an FPGA-based BiCAM was designed and used to store its mnemonics instruction sets rather than the used RAM.To verify the efficiency of the proposed method, a comparison with common RAM-based search operations is presented. Time complexity was chosen as the analysis method to examine the results, as we are concerned with the speed of the system. As expected, the time complexity of the proposed BiCAM was found to be a constant value of O(1) under all cases, regardless of the size of the input. In contrast, the time complexity of RAM-based search operations varies. To sum up, this thesis introduces a hardware-based enhancement in the form of BiCAM to be used in assemblers as a storage unit to speed up their conversion process.

Keywords: Content Addressable Memory (CAM), Field-programmable Gate Array (FPGA), Assembler, Mnemonics Instruction, Search Operation, Time Complexity.

İÇERİK ADRESLENEBİLİR BELLEK (CAM) KULLANILARAK FPGA TABANLI ASSEMBLER TASARIMI VE UYGULAMASI

ÖZET

Bir assembler, bir programın işlemci tarafından icra edilebilmesi için makine koduna dönüştürülmesi işlemini sağlayan bir araçtır. Dönüştürme işleminde bellekte yer alan işlemci komut kümesinden yararlanılır. Bundan dolayı programdaki herbir komutun ikili karşılığı komut kümesinde arama işlemi yapılarak elde edilir. Arama işleminde doğrusal, ikili ve karma tabanlı algoritmalar ana türlerdendir. Sistemin toplam verimliliğini artırabilecek ve kritik bir faktör olarak kabul edilebilecek arama işlemini hızlandırmak için iyileştirmeler yapılmaktadır. Literatürde yer alan arama algoritmalarının ortak yönü donanım tabanlıdan ziyade yazılım tabanlı yöntemler kullanmasıdır. Bu tezde, ikili kodun elde edilmesindeki arama işlemini hızlandırmak donanım tabanlı bir yaklaşım getirildi. Önerilen donanım tabanlı yaklaşım, yaygın olarak kullanılan Rastgele Erişim Belleği(RAM)'in yerine İkili İçerik Adreslenebilir Bellek(BiCAM) kullanılması ile gerçekleştirilmiştir. BiCAM, arama işlemini tek bir saat döngüsünde gerçekleştirebilen donanım tabanlı bir paralel bellektir. FPGA tabanlı BZK.SAU assembler, eğitim amaçlı bir assembler olup bu tezde önerilen yaklaşımı değerlendirmek için vaka çalışması olarak seçilmiştir. BZK.SAU Assembler'in dönüşüm işleminde kullandığı komut kümesi için FPGA tabanlı bir BiCAM tasarlanmış ve kullanılmıştır. Önerilen yöntemin etkinliğini ölçmek için, ortak RAM tabanlı arama işlemleriyle bir karşılaştırma sunulmuştur. Zaman karmaşıklığı, sistemin çalışma hızıyla ilgili bilgiler verdiğinden dolayı seçildi. Beklendiği gibi, önerilen BiCAM'nin zaman karmaşıklığı, bellek elemanının boyutu ne olursa olsun tüm durumlarda O(1) olan sabit bir değer olarak bulundu. Diğer RAM tabanlı arama işlemlerinin zaman karmaşıklığı ile karşılaştırıldığında önerilen yöntemin etkinliği göstermektedir. Özetle, bu tez geleneksel bellek yerine BiCAM'ın kullanılması assembler'in ikili koda dönüştürme süresini hızlandıran donanım tabanlı bir yaklaşım getirmektedir.

Anahtar Kelimeler: İçerik Adreslenebilir Bellek (CAM), Alanda Programlanabilir Kapı Dizisi (FPGA), Derleyici, Mnemotekni Talimatı, Arama İşlemi, Zaman karmaşıklığı.

CHAPTER 1. INTRODUCTION

Assembly language (ASM) is a low-level programming language that is used for computers or any programmable device to convert the codes written by programmers (Source Code) into machine language (stream of zeroes and ones) [1,2]. As the machine language is difficult to understand by human being, Assembly Language helps programmers to write their codes by using mnemonic representation (English syntax). Where sometimes it is called symbolic machine code. In contrast to most high-level programming languages which generally can be used by multiple architectures but just need to be interpreted or compiled [2], each assembly language is specific to a particular computer architecture. That is why we see each manufacturer use its own assembler to convert their assembly languages to the machine language.

An assembler is a software program that is used to convert the assembly language into an executable machine code to perform the processor's basic operations [3]. Where the word assembly refers to the conversion process. Sometimes, an assembler is called a compiler for assembly language as it provides interpreting services as well. The assembler can be considered as a connection point between the assembly language (which is symbolically coded) and the computer processor, memory, and the other computational elements. Generally, assemblers are categorized as single-pass and multi-pass based on how many times they read the source code before translating them into object code (machine code) [4]. The one-pass assemblers read the source code only once, but multi-pass assemblers do it several times, creating a table of all the symbols and their values in the first pass.Then, the table is used to generate the code in the subsequent passes. Assemblers store their mnemonics and their corresponding machine language in memory to be extracted using a search operation and implemented by the Central Processing Unit (CPU) or microprocessor.

Memory is an essential component of computing systems, as it is used to store and retrieve data and instructions. Generally, the computing systems count on both memory and the CPU to perform their operations [5]. where memories help the CPU to run the

computing systems by storing the needed instructions to perform tasks. [6]. In all computing systems starting from simple embedded systems to super-computers, memories take an important role in determining the factors of the system efficiency such as power consumption and system speed beside the CPU [7-9].

Modern applications have a growing requirement for large data storage and quick access to it, which drives the need to develop computer memory technologies that go beyond what is possible with conventional memory. Computer systems frequently use a type of conventional memory called Random-access memory (RAM). It provides non-permanent storage space and a location for accessing data, allowing applications to fast access the information they need [10]. Accessing specific data or instructions in memory requires performing a search operation. There are various types of search operation techniques and algorithms that are designed to enable quick and accurate access to data and instructions. Linear, Binary, and Hashing-based algorithms are the main search techniques used [11]. Enhancing the search operation helps to improve the system's efficiency by speeding up the process of picking up the required instruction that will be executed by the CPU from the memeory.

In recent decades, technology has experienced rapid development and the adoption of new methods and approaches. As a result, the need to have a faster computing system with higher efficiency has increased. The use of Content-Addressable Memory (CAM) which is also known as Associative Memory instead of conventional memories can meet this need. CAMs are a special type of memory used as a hardware-based search engine that is composed of conventional semiconductor memory (usually Static Random-access Memory SRAM) with added parallel comparison circuitry that simultaneously searches all memory entries in a single clock cycle [12]. The concept of content-based memory is derived from the biological brain, in which stored data is accessed regardless of its location [13,14]. In contrast to RAM, which is also known as address-based memory which needs indexing of data to be accessed or stored [15]. CAM uses the same principle as the human brain where it stores information and does not need any index to retrieve the information, where only the content itself or a portion of it is used to retrieve the searched-for information.

Field Programmable Gate Arrays which are abbreviated as FPGAs are integrated circuits that are made up of a matrix of configurable logic blocks (CLBs) linked to each other by programmable interconnects [16]. They commenced appearing in the digital circuit industry in 1985 and have since become a widely used technology [17]. Due to their feature of reconfigurability (flexible devices), FPGAs can be programmed to design and implement any digital circuits from high-level down to transistor-level architectures. FPGAs have the ability of bit-level reconfiguration which enables the exact design of hardware for each specific application in contrast to a fixed hardware design. The benefit of having such a highly programmable circuit is to minimize the used area and increase the speed by decreasing the consumed power.Generally, FPGAs are made up of Configurable Logic Blocks (CLBs) that can be utilized for implementing the logic functions, a programmable interconnect to make the connection between CLBs, and the Input-output blocks to communicate between the internal parts of the FPGAs and the outer world [18].The FPGAs can be programmed to the desired functionality by using one of the Hardware Description Languages (HDL) such as Verilog , Very High-Speed Integrated Circuit (VHSIC) hardware description language (VHDL) [19]. FPGAs are commonly utilized in numerous applications because of their ability to be programmed and operate in parallel. These applications include Aerospace & Defense, High-Performance Computing and Data Storage, Wireless Communications, Video & Image Processing, etc. [20].

BZK.SAU.FPGA is an FPGA-based microcomputer architecture designed and implemented on Altera's Cyclone II Development board by using a computer architecture simulator known as BZK.SAU [21]. BZK.SAU is a computer architecture simulator that all its units were designed at the logic gate level and has the instruction sets that are generally used in commercial microprocessors. The BZK.SAU.FPGA is the case study of this thesis, and it will be utilized for evaluating and testing the proposed approach. The BZK.SAU was mainly designed as an educational tool forwarded to undergraduate students to improve their understanding of computer science courses. This microcomputer architecture has been used by Sakarya University Department of Computer Engineering Since 2009 [21]. The use of BZK.SAU helps students to go from theory to practice through its ability to reconfigure the circuit which allows students to perform unlimited number of design iterations at no additional cost. In addition, it has a adjustable clock feature, which enables students to monitor the execution of the

program code and check the status of the registers at any desired time. The BZK.SAU has a special assembler that was created specifically for it. The brute-force algorithm was employed as a search operation to convert the source code into the machine code [22]. The BZK.SAU.FPGA is designed with a modular approach, allowing users to easily add additional units, such as addition or subtraction units, without the need to completely redesign the system from scratch. Also, the system's operations can also be observed step by step.

The BZK.SAU.FPGA has a range of features, including the ability to support six different addressing modes (immediate, direct, indirect, indexed, relative, and inherent), 16-bit data and address buses, and a set of 8 registers: the Address Reg. (AR), Data Reg. (DR), Program Counter (PC), Accumulator (AC), Stack Pointer (SP), Index (IX), Temporary Reg. (TR), and the 8-bit Instruction (IR) Reg. . It also has a unit for calculating effective addresses and a set of 51 instructions, which are divided into 21 for memory and accumulator operations, 8 for index and stack register operations, and 22 for branching [23]. Like many other programmable devices that use assembly language, the BZK.SAU.FPGA relies on a search operation within its memory (RAM in our case) to convert mnemonics instruction sets into machine code for internal operations and communication with the outer world. However, this search operation can be time-consuming due to the limited hardware structure of the RAM, which can be problematic in applications that are time sensitive. To solve this, it is suggested that the RAM be replaced with an FPGA-based BiCAM as a storage unit to improve efficiency and speed.

In summary, while several approaches have been proposed to accelerate the search operation within memories, they tend to rely on software-based techniques that do not modify the underlying hardware structure of the memory. This can be limiting due to the inherent constraints of memory hardware structures. To address this issue, this thesis proposes a novel approach that uses a hardware-based technique called FPGA- based BiCAM to enhance the efficiency of the search operation within memories. The BZK.SAU.FPGA assembler is used as a case study to demonstrate the effectiveness of the proposed approach, in which the BiCAM is implemented as the memory structure instead of the traditional RAM.

To assess the proposed approach, a RAM-based Brute-force search algorithm is implemented as a linear search technique to make a comparison with the suggested approach. The Time complexity of the proposed approach is also compared with the time complexity of other common RAM-based search algorithms. This allows for a thorough assessment of the proposed approach's effectiveness and efficiency. Time complexity is a measure of the efficiency of a search algorithm, which determines how long an algorithm takes to run based on the length of its input. It calculates the time needed to execute each coding command in the algorithm. The time complexity of the proposed approach is expected to have a fixed value of O(1), meaning that the search operation is completed in a single clock cycle regardless of the size of the search value or memory. While the proposed approach may come with an additional cost due to its reliance on hardware, it is still considered a good trade-off in situations where a time-sensitive application is needed, and it does not need a large memory.

The remainder of this thesis is organized as follows: Chapter 2 presents a summary of the architecture of the BZK.SAU.FPGA microcomputer and its organization, along with an explanation of its instruction sets and addressing modes, and an example of an assembly code for the BZK.SAU.FPGA assembler. Chapter 3 provides a general overview of Content-Addressable Memory (CAM) circuits, including their working principles, types, and architectures, as well as a summary of their advantages and disadvantages. Chapter 4 discusses FPGA-based CAMs, including an introduction to FPGAs and their components, mentioning the main manufacturers of FPGA boards, and a description of the various types of memory that can be used in an FPGA to create a BiCAM. Chapter 5 explains the methodology of this work, which is divided into two subsections. The first one covers the proposed FPGA-based BiCAM design in detail, and the second one discusses the evaluation of the proposed method by implementing a search operation using the proposed BiCAM and a RAM-based Brute-force search algorithm. Chapter 6 presents the results of the evaluation and discusses the time complexity analysis of the proposed approach and the main RAM-based search algorithm. Finally, Chapter 7 provides the conclusion of this thesis.

CHAPTER 2. CASE OF STUDY: BZK.SAU.FPGA

2.1. Introduction to BZK.SAU.FPGA

BZK.SAU.FPGA is a 16-bit microcomputer designed in the FPGA development environment with a modular logic gate architecture [21,24]. Altera Cyclone II development board with EP2C35F627CNC chip was the chosen board for the design whereas Altera Quartus II was the software used for the implementation [21]. The BZK.SAU is a computer architecture simulator that is used to design architectural units at the logic gate level with the commonly used instructions in the commercial microprocessor [23]. The BZK.SAU.FPGA is primarily intended to be a microcomputer designed as an educational tool for college students to enhance their understanding of the computer science department courses. In 2009 it was first used by the Computer Engineering Department of Sakarya University [21]. The BZK.SAU.FPGA has a modular design in the sense that any additional units that may be required in the future, such as adding, subtraction, and so on could be simply added to the user's design rather than redesigning the program from scratch without causing any negativity in the operation of the system. Also, the system's operations could be monitored gradually [25,26]. The modularity feature of the BZK.SAU.FPGA helps the student to switch from a traditional, theory-based learning approach to a more practical approach by adding more hands-on activities to courses. By reconfiguring the architecture, students can perform an unlimited number of design repetitions without incurring additional costs. Additionally, the BZK.SAU.FPGA allows students to witness the execution of program codes and view the status of the registers at any moment. The BZK.SAU.FPGA has a custom assembler that uses the Brute-force as a search algorithm to convert user source code into machine code, and includes an instruction set of 59 instructions that support six different addressing modes [24-26]. An overview of

the BZK.SAU.FPGA is provided in Table 2.1, and more detailed information about its architecture is discussed in this chapter.

Table 2.1 : BZK.SAU.FPGA Microcomputer architecture design summary.

Feature	Explanation
System Name	BZK.SAU.FPGA
System Hardware	FPGA (Altera DE2-70)
Output unit	VGA Monitor (640×480)
Screen size	40 columns×24 rows (320×384)
Input unit	PS/2 keyboard
Description Language	Schematic (Hardware)
Processor Architecture	Von-Neumann (SISD structure)
Processor Type	16-bit
Address Path	16-bit
Bus width	16-bit
System Registers	10 pcs (Input and output registers 8-bit others 16-bit wide)
Main Memory	64 KB – 16 bits
Secondary Memory	Flash Memory (4 MB) – 8 bits
Memory Layout	Big-Endian
Command Architecture	CISC
Command Set	Functional, Control, Transfer, Input-Output, and Stack Commands (59commandsd)
Instruction Structure	16 bits (bits 15-12 addressing mode, bits 11-0 opcode field)
Command Processing Method	None-Pipeline
Addressing Mode Type	6(Immediate, index, direct, relative, indirect, and inherent)
Control Unit Structure	Hardware
ALU Unit	16-bit (Integers only)
Number System	2's Complement
Operating System	Single User-Single Task (BZK.SAU.OS)
File System	FAT-16
Assembly Language	BZK.SAU.ASM Assembly Language

2.2. Architecture of BZK.SAU.FPGA

The BZK.SAU.FPGA microcomputer architecture uses a Video Graphics Array VGA type for display with a resolution of 640x480 pixels. A 40×24 screen has been used since the memory size of the architecture is limited and by considering the used screen size in other systems such as ABC80 [27] which is 40 columns×24 rows too. The PS/2 keyboard is preferred as the input unit in the system architecture as it has a simple communication method. It allows the use of all keys except the function keys found on a conventional 102-key PS/2 keyboard. The American Standard Code for Information Interchange ASCII code table in the literature was used in the process of assigning codes to the keys. The system architecture has been addressed and a bus width of 16-bit with a main memory unit capacity of 64KB is used. We can say that most of the system design was made schematically. That means it was designed at a big-endian logic gate level. FPGA is the hardware unit that the system was built on. Quartus development tool was used for programming and converting logic-level designs into software languages by using VHDL. Due to the general structure and the educational purpose of the system, a SISD architecture based on a single instruction-single data basis is used for simplicity. Flash memory was used as a storage unit in this architecture due to its availability in existing resources and ease of control. big-endian was used as a memory layout in both the main memory and storage unit. Furthermore, as the number of microprocessing steps of the commands used differs, the microcomputer architecture commands are in the complex instruction set computer CISC structure. In Figure 2.1 the block diagram of BZK.SAU.FPGA microcomputer architecture is represented.

There are 10 general and special purpose registers used by the architecture of which the input and output registers are 8-bit, and the others are 16-bit wide. Also, it has six basic addressing modes, which are widely used in both educational computer architectures and commercial microprocessor instruction sets in literature. They are as follows: inherent, immediate, direct, indirect, index, and relative. Hardware control is used in the microprocessing steps of the command structure. While the architecture's arithmetic and the logic unit can only operate on 16-bit integers, it employs 2's complement logic, it is widely employed in literature to represent negative numbers.

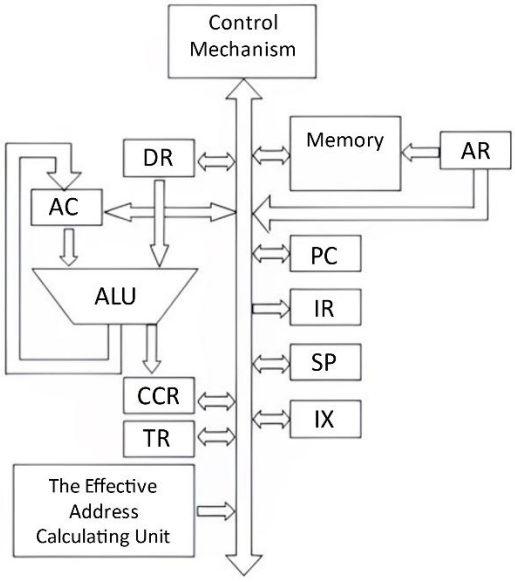


Figure 2.1 : The BZK.SAU.FPGA Microcomputer architecture block diagram.

The basic operating system commands operating on the microcomputer architecture by utilizing the assembly instructions of the architecture. Where it allows the user to write a new program, compile and run them, store them in the storage unit, and edit them by calling from the storage unit. The file layout in the storage unit is based on the FAT-16 file system, which is the basic file system and serves as the basis for other file systems. For more details of this architecture, references [25,26] can be checked.

2.3. Memory Mapping and Organization

In the microcomputer architecture, memory configurations and memory maps were created to facilitate operating system and assembler research as in Figures 2.2 and 2.3.

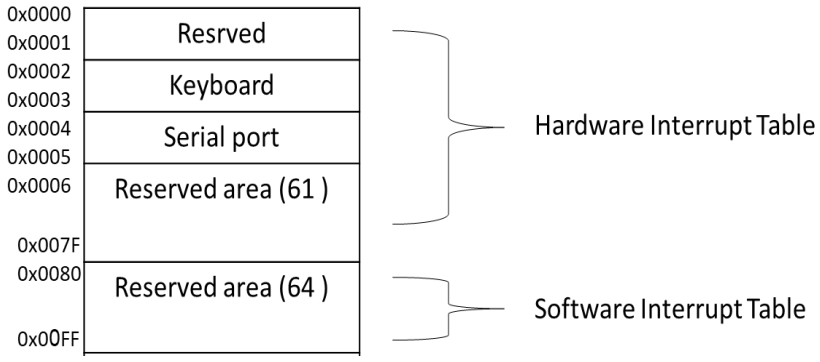


Figure 2.2 : Memory mapping part-1.

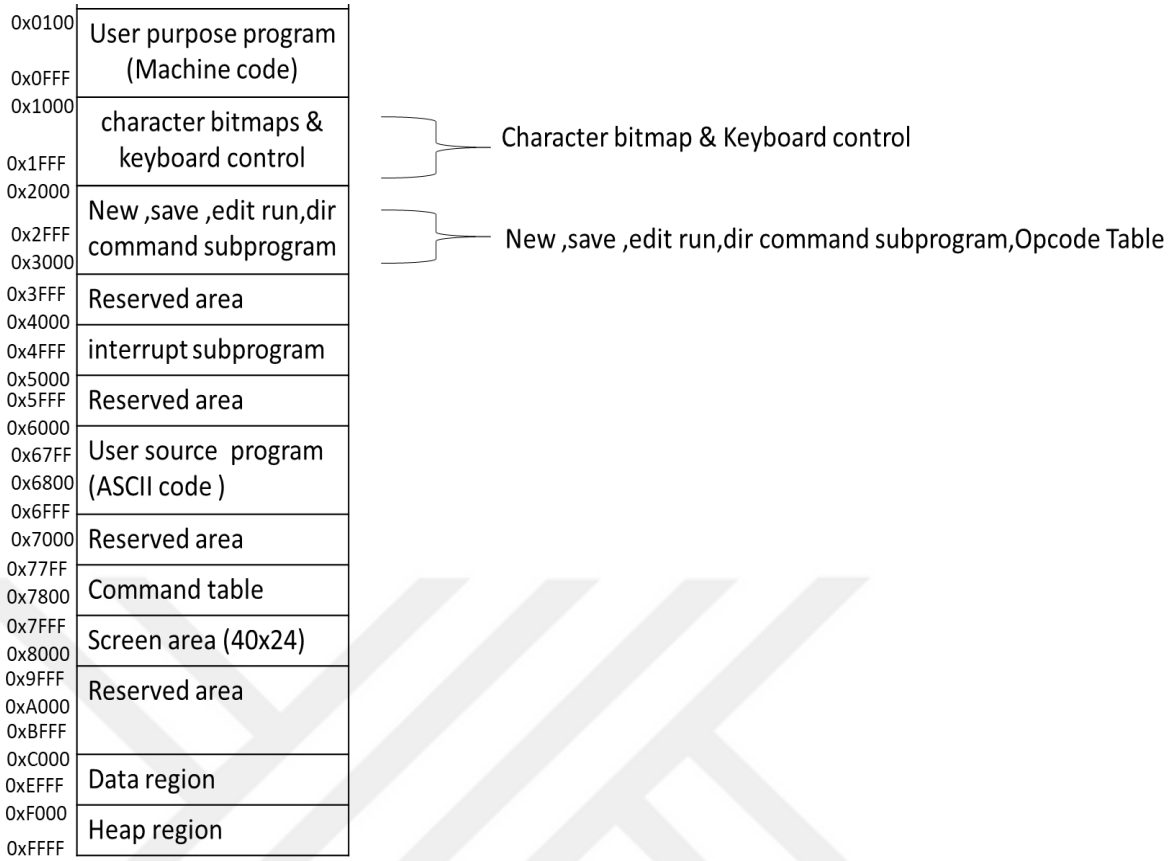


Figure 2.3 : Memory mapping part-2.

From Figures 2.2 and 2.3 we can see that the first 256 bytes of memory are held for interrupt tables, both hardware, and software. Each interrupt is assigned to 2 bytes. The total number of reserved interrupts is 128. When the user source program (ASCII code) is converted to machine code, 1k of memory is reserved for the Assembler program that allows the user program (machine code) to be transferred to its reserved area.

2.4. Boot Process

When the BZK.SAU.FPGA microcomputer starts up it first executes the boot code from the 23-byte M4K type memory and places the following data in the 64k memory unit as given in Figure 2.4:

I. Assembler program.

II. Keyboard Control program.

III. Command Table.

IV. Character bitmaps.

V. Cutting Charts.

After replacing this data, the program flow of the microcomputer is transferred to the main program. The M4K type memory is a type of memory that can be used as RAM or ROM which is available on an FPGA board [22]. This kind of memory could be loaded up by a startup file with (.hex) extension via a user interface that is developed by the vendors of the FPGA cards. Due to that the fixed data and the boot code of the microprocessor could be initially replaced in the main memory by loading the two files with. hex extension into the memory of 8k and 23 bytes respectively. After that, the subprogram finishes its execution, it branches into the subprogram that makes the cursor active/inactive at regular intervals on the screen. The assembly instructions of this subprogram are in the APPENDIX-C of the reference [25].

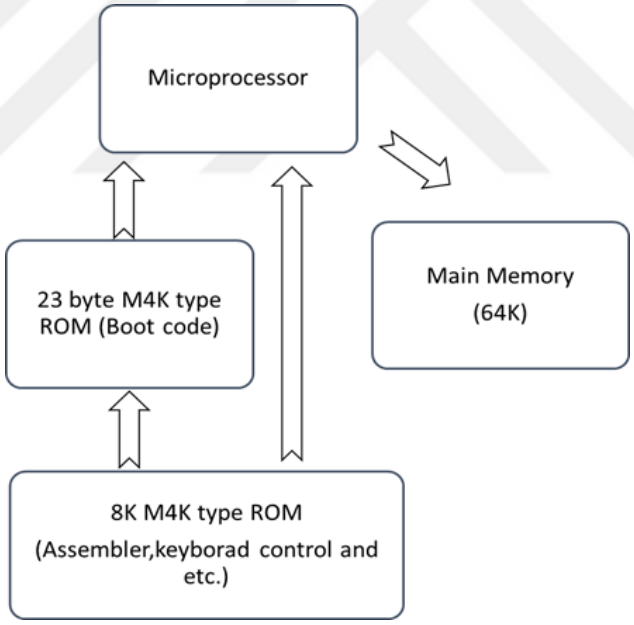


Figure 2.4 : The flowchart of the boot process.

2.5. BZK.SAU.FPGA Assembler

The microcomputer unit of the BZK.SAU.FPGA is made up of 59 instructions that are divided up as given in Table 2.2 below with six distinct addressing modes. Each specific instruction has a 16-bit relevant machine code. The addressing modes and their corresponding symbols of the assembly language that is used for the microcomputer are shown in Table 2.3 below. As we can notice from the inherent addressing mode occupies no words in memory as it has no related symbol, while the other addressing modes occupy one word in memory as it contains it related symbol.

Table 2.2 : Distribution of instructions in BZK.SAU.FPGA.

Type	Number of instructions
Memory & Accumulator	21
Index & Stack	8
Unconditional & Conditional Branching	22
Condition Code Register	6
Input-Output	2

Table 2.3 : Addressing modes and their symbols that used in BZK.SAU.FPGA.

Addressing Mode Type	Symbol
Immediate	#
Direct	$
Indirect	@
Index	%
Relative	*
Inherent	

The user could write a program on the 24x40 lines screen using assembly commands if the following rules were followed. A space must be left after the assembly language command. If a command using the inherent addressing mode is to be used, no spaces should be left, and the assembly language command must be terminated with the

semicolon symbol “;”. After the space character the symbol of which addressing mode will be used should take a place as the addressing mode symbols given in Table 2.3. The data to be used after these addressing mode symbols should be in hexadecimal format, with the character "h" appended to the end of it. The semicolon character “;” should be used at the end of each command. After that, an explanation of the command could be written. Each new command needs to be entered on a new line. Therefore, each new line needs to enter with the "Enter" key. At the end of the program, the "-r" directive is used at the end of the program to switch the written program into machine code. Figure 2.5 contains an example of an assembly program used for BZK.SAU.FPGA.

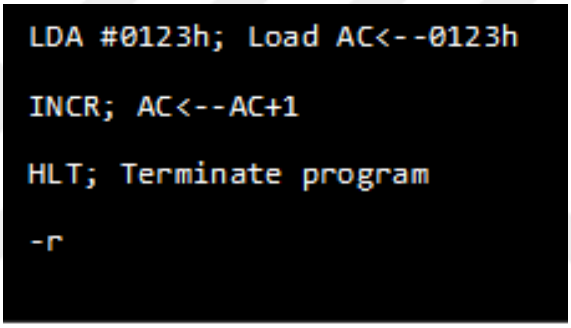


Figure 2.5 : A BZK.SAU.FPGA assembly code example.

CHAPTER 3. CONTENT-ADDRESSABLE MEMORY (CAM): CIRCUIT AND ARCHITECTURE

3.1. Introduction to Content-Addressable Memory (CAM)

Content-Addressable Memory, also known as Associative Memory, is a type of Memory that is used to store data and retrieve it by implementing a lookup-table function in a one-clock cycle using devoted comparison circuitry [13]. Also, it can be explained as a hardware-based search engine [28]. CAMs' search operation is implemented by making comparisons between its contents that are stored in a table with a key (searched value) instead of finding the address as the traditional memories where the search process is done sequentially and takes multiple clock cycles to find the desired value.

This method makes the search process faster than software-based search operations [13]. Also, it has a parallel active circuitry that enables searching within its entire contents just in one clock cycle [29]. As the parallelism property of CAMs improves its search process, CAM searches all memory cells in one clock cycle for the search value using a given key and returns the matching contents via a parallel active circuit [13]. But the use of parallel active circuitry may lead to an increase in power dissipation and hardware cost. To solve these, various energy-saving designs and technologies for CAM are suggested in the literature [28,30-35]. Despite the disadvantages of CAMs that have been mentioned. CAMs are still used rather than traditional memories for rapid, data-intensive, and time-sensitive applications due to their excellent throughput and fast response. As a result, they are considered an adequate trade-off and are widely used in multiple of applications, such as packet forwarding and packet classification in network routers, database, data compression, Artificial neural networks, network security, and pattern recognition and matching [36- 42,102].

There are two kinds of CAMs, the first one is Binary content addressable memory (BiCAM), in which each memory unit cell of it can have either a '0' or a '1' state, and the second one is Ternary content addressable memory (TCAM), in which it has the same cell states as the BiCAM in addition to one more state which is 'don't care ('X') state [43,44]. The BiCAM type is chosen for this thesis as we are only interested in the '0' or '1' state not in the don't care state. Also, CAMs are classified into two categories based on their implementation technique: Application-specific integrated Circuit (ASIC)-based CAM (traditional) and field-programmable gate array (FPGA)-based CAM [45]. FPGA devices, as opposed to ASICs, provide a useful platform for digital system designers due to their high flexibility and reconfigurability. but unfortunately, the existing FPGA architectures don't provide a CAM block unit. So, several methods of FPGA-based CAM design are presented in the literature. These methods include Block RAM (BRAM)-based CAMs [46,47], Distributed RAM-based CAMs [48], and Flip-flop-based CAMs [49].

3.2. RAM vs. CAM

In contrast to the conventional computer memory RAM, where usually the user provides the memory address and the RAM reverts the data stored at that specific address, a CAM is set up in a way that the user provides the data by itself, and the CAM looks up through a table of the stored data (memory array) to check if desired data is stored anywhere in it or not. In general, RAM is used to read and write data as it is one of the primary memories of computing systems. Where it is used to store the needed data to perform system or program operations. Two kinds of RAM exist: dynamic RAM (or DRAM) and static RAM (or SRAM) [50]. CAMs are also used for reading and writing data and can take a part of the primary memory which is used for fast-paced applications where the search process must be done faster. The RAM-based research operation involves sequentially scanning through the memory locations until the desired value is found. This can be a relatively slow process, especially if the value is not located near the beginning of the memory. on the other hand, The CAM-based search operation is carried out in parallel where it simultaneously compares the desired value to all the values stored in the memory. If a match occurs, the match line will indicate the location of the match. As a result, multiple clock cycles are needed to find the required

data in the RAM-based search operation as opposed to the CAM. where only one clock cycle is required. In Table 3.1 the main differences between RAM and CAM are summarized.

Table 3.1 : The main differences between RAM and CAM.

Memory Type	RAM	CAM
Application	Used to run programs and store their needed data while they are running	Usually used in database management systems.
Suitability	Suited to the sequential search algorithm	Suited to parallel access
Operation	An address is provided, and RAM retrieves the data located at that address.	A search data is provided, and the CAM returns the address of the found data.
Cost	Cheaper than CAMs	More expensive than RAM

In Figure 3.1 an example of a search operation is given for both RAM and CAM to have a clearer explanation of the difference between them.

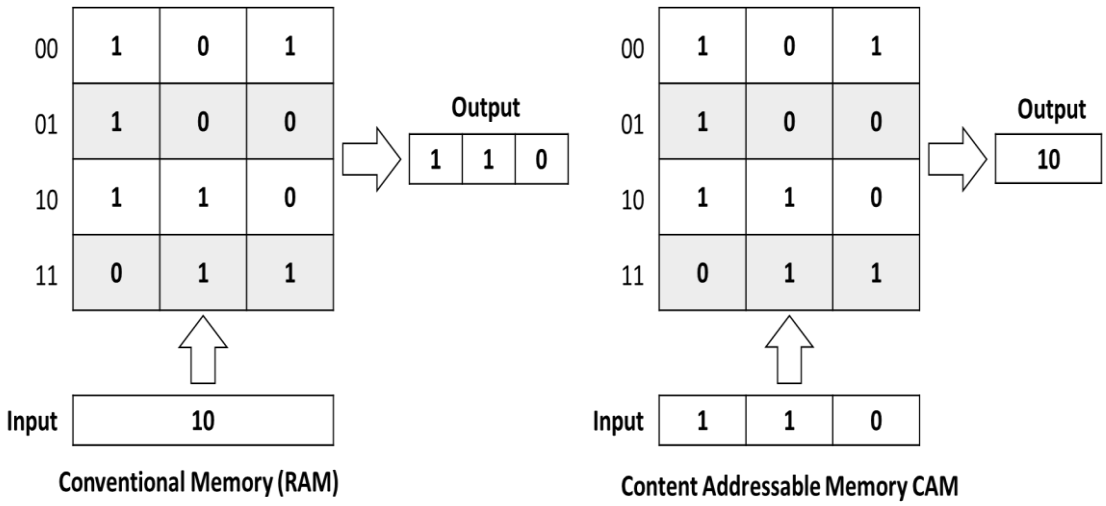


Figure 3.1 : RAM versus CAM in reading operation.

3.3. Organization of CAMs

Generally, CAMs are consisted of 4 main components as given below:

- **Argument Register** (Arg_reg): It contains the data that will be searched for. It is composed of n bits.
- **Key Register** (Key_reg): This register has n bits that specify which portion (bits) of the argument register should be compared with the memory array. If the bits set to "1", the corresponding bit should be compared. Otherwise, the bit is ignored.
- **Associative Memory Array**: It is a nxm bits array that includes the data to be compared with the argument register.
- **Match Register**(M): It is made up of m bits, one for each data (row) of the memory array. Following the matching process, the associated bits are set to 1 after a match happens.

The block diagram of CAM is given in Figure 3.2.

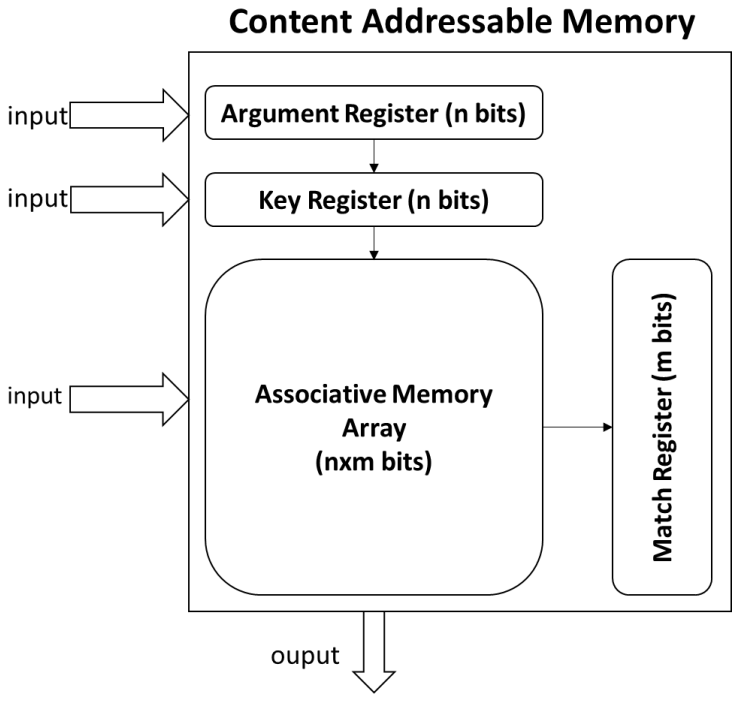


Figure 3.2 : CAM Block diagram.

3.4. Architecture of CAMs

3.4.1. General view

The CAM structure can be implemented using either a NAND gate or a NOR gate at the circuit level. At the architecture level, it uses SRAM cells for bit storage and either an XOR or XNOR logic gate for comparison operations [28,43,51]. In Figure 3.3, the general architecture with necessary drivers and peripherals of the CAM is shown.

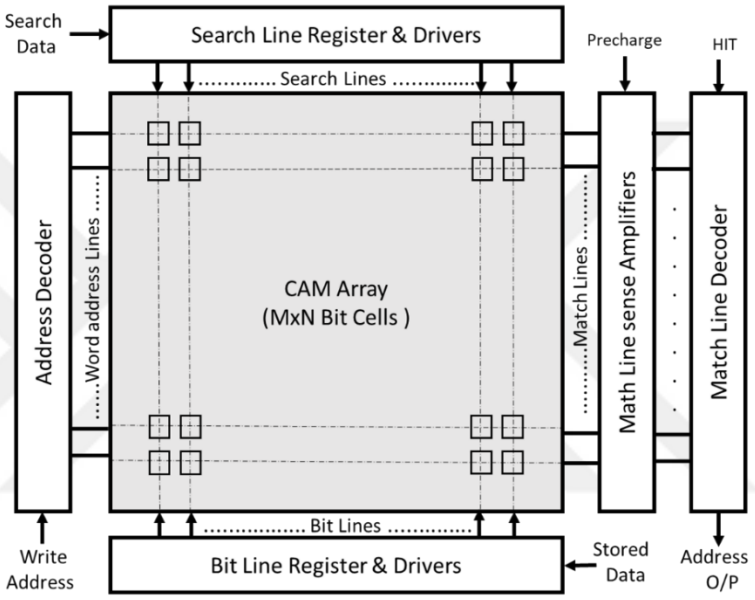


Figure 3.3 : Architecture with necessary drivers and peripherals of the CAM.

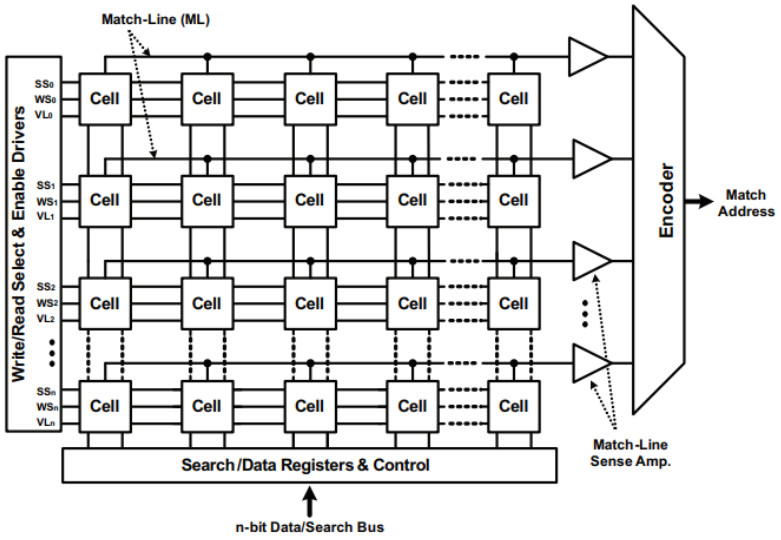


Figure 3.4 : Conventional CAM architecture.

In Figure 3.4, a block diagram of an example of a conventional 4-word CAM architecture is given, where each word consists of 4 bits, and each bit is stored in a cell where it will later form a row of bit cells. The search operation is implemented by loading the search data bits from the key register to the comparison circuit inside each CAM bit cell via the search lines. Each row has its match that is connected serially to all the bit cells of the same row. which will have the result of the comparison operation whether a match happens or not. in addition, each match line is connected to the match line sense amplifier.

3.4.2. Core bit cells

A CAM cell basically performs two fundamental tasks: bit storing (as in RAM) and bit comparison (unique to CAM). SRAM is most often used for storage in CAMs as given in Figure 3.5, where single-bit cell storage is implemented by cross-coupled inverters that implement the bit-storage nodes Q, and $\overline{Q}$.

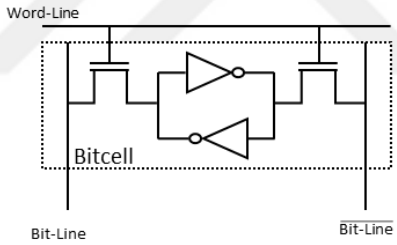


Figure 3.5 : SRAM bit cell.

For each bit cell, two access nodes known as bit-lines (BL) exist. These bit lines contain the value of the stored bit and its complement. On the bit-lines, two access transistors enable and disable access to the stored data for reading and writing operations. where the signal that controls the access transistors is known as a word line (WL).

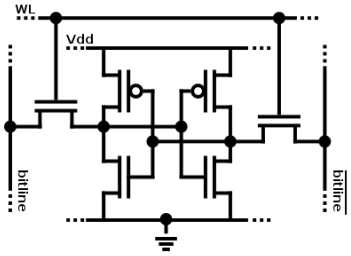


Figure 3.6 : Most common 6T SRAM bit-cell.

The most basic SRAM bit cell is the 6-transistor bit cell (6T cell) as given in Figure 3.6. This cell consists of two weakly cross-coupled CMOS inverters and two access transistors which are used to read and write the stored data [52,53]. The bit cell comparison operation between the search bit and the stored is logically equivalent to using an XOR bitwise operation.

3.4.3. Types of CAM bit cells and their comparison techniques

There are three main types of CAM bite cells: NOR, NAND, and Ternary Cells. [13]. Each of the bite cells is explained in this subsection in detail.

3.4.3.1. NOR-Cell

The complement of the stored bit is compared with the complement of the relevant search data that is stored in the search line after NOR cells have been employed. 4 transistors (to keep high cell density) designed to implement a pull-down of an XNOR gate with searchline and stored bit, with each pair creating a pulldown path from the match-line by connecting the match-line to ground. When the search line and stored bit match, both pulldown paths are disabled, and the match line is disconnected from the ground. In Figure 3.7 a 10-T NOR-type CAM core cell is shown.

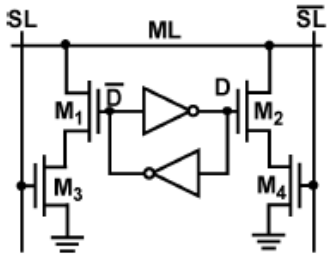


Figure 3.7 : CAM Core cell for 10-T NOR-Type.

3.4.3.2. NAND-Cell

The NAND cell implementation differs from the NOR when comparing the stored bit and corresponding search data. Where it uses 3 comparison transistors instead of 4. This cell implementation becomes clear when connecting multiple NAND cells serially, see Figure 3.8. As the pulldown path of a CMOS NAND gate is made by a series of nMOS,

the connected nodes will form a row in this condition. To satisfy the match condition for an entire row every cell in the row must get matched.

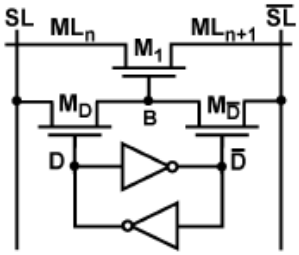


Figure 3.8 : CAM Core cells for 9-T NAND -Type.

3.4.3.3. Ternary cells

The previous NOR and NAND cells are used as binary CAM cells which can hold only either a “0" state or a "1" state. the Ternary cells in addition to "0" and "1” can store an "X" state. The "X" state is a don't care condition, which means that storing an "X" state in a cell will result in a match regardless of the corresponding search data. To store a ternary value in a NOR cell, an additional SRAM cell is added. The pulldown paths can be handled independently thanks to the first node's connection to the left pulldown path and a second node's connection to the right pulldown path. By not setting both nodes to logic "1," an “X," state is stored. This makes the cell match regardless of the search data and disables both pulldown paths. An illustration of this is given as an example in Table 3.2 and Figure 3.9.

Table 3.2 : Ternary encoding for NOR gate.

	Stored Bit		Search Bit	
Stored Value	D	$\overline{D}$	Stored Value	D
x	1	1	0	0
0	0	1	0	0
1	1	0	1	1

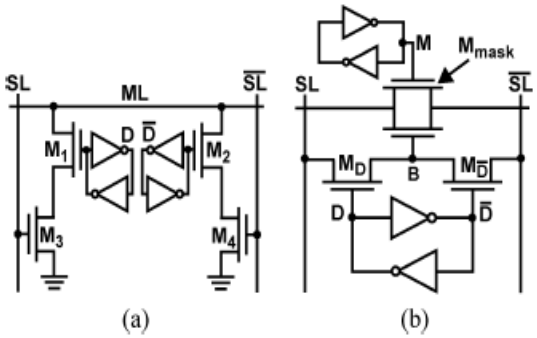


Figure 3.9 : CAM Ternary core cell for ; (a) NOR-Type & (b) NAND-Type.

3.4.4. Matchline structure

The match lines indicate if a match happens between the stored data and the search data or not. The address of the matched data is determined based on the value of the match lines (0 or 1) that are connected to an encoder. where the used encoder is assumed to have only one match per time. Furthermore, frequently there is a flag that is responsible for indicating the condition of no matching in the CAM is found. Each row of the CAM has its match line that is linked to match-line sense amplifiers (MLSAs). The match line is one of the most important structures in CAMs. Usually, CAMs match lines are built using NOR and NAND cells [54].

3.4.4.1. NOR Matchline

A NOR match line is formed by connecting the NOR cells in parallel. It works like that as the NOR-based search operation is composed of three stages. The first one is the search line pre-charging by disabling the pull-down paths in each CAM cell. After that, the search lines are precharged low to disconnect the match lines from the ground. Then in the second one, which is the match line pre-charging the M pre-transistor precharges the match lines high when the pull-down paths are disconnected. Finally, in the last stage, which is match line evaluation if a matching case happens, the match line voltage remains high as there is no discharge path to the ground. The match line sense amplifier (MLSA) detects voltage on match lines and produces an output as the match result. The NOR match line's key advantage is its high speed. In Figure 3.10 the structure of a nor match line with n cells transistor is shown.

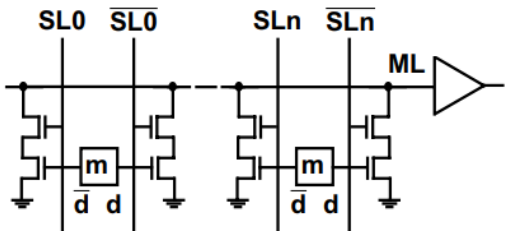


Figure 3.10 : NOR-Type matchline structure.

3.4.4.2. NAND Matchline

In the NAND match line, if there is a match, all the nMOS transistors (M1...Mn) are turned on, effectively causing the match line to discharge to the ground from the match line nodes. In the case of a match, all nMOS transistors, M1 through Mn are on, effectively creating a path to the ground from the ML node, hence discharging ML to the ground. In the case of a mismatch, at least one of the nMOS transistors chain (M1... Mn) is turned off leading to a high match line voltage. The difference between the match (low) voltage and the miss (high) voltage is found using a sensing amplifier. Unlike the NOR match line, which is evaluated by the CAM cells, the NAND match line has an explicit evaluation transistor. In Figure 3.11 a NAND matchline structure with pre-charge and evaluate transistor is presented.

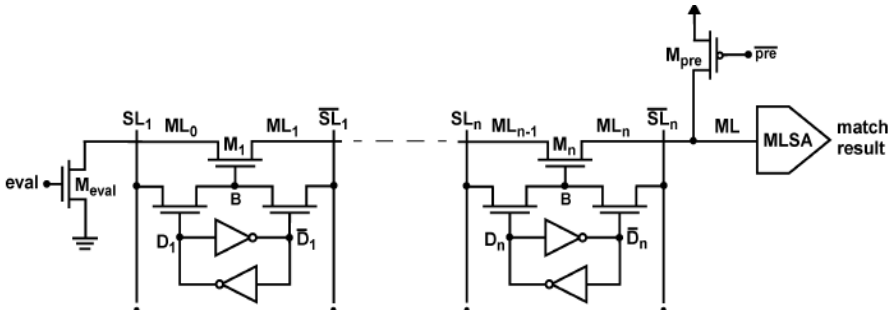


Figure 3.11 : NAND-Type matchline structure.

3.4.5. Matchline sensing schemes

3.4.5.1. Conventional (Pre-charge-High) matchline sensing

The basic principle of sensing a NOR-type match line is to charge the match line high and then evaluate it by letting the NOR cell discharge their matchlines in case of a

mismatch or keep the match line high in the match case [55]. In Figure 3.12 the schematic of pre-charge match line circuitry is shown.

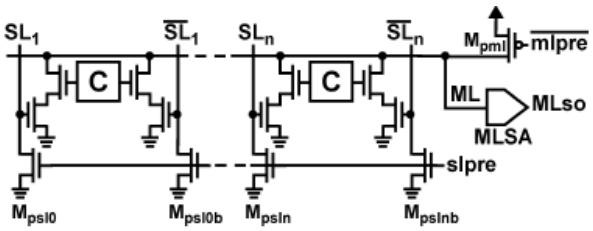


Figure 3.12 : Pre-charge matchline sensing circuitry.

3.4.5.2. Selective-precharge scheme

In selective pre-charge, the first bunch of data bits is should be matched before the beginning of the search operation for the remaining bits. where it is the most used way to save power on match lines.in figure 3.13 a sample implementation of a selective Pre-charge match line is given. where we can observe that the are three cascade CAM cells where the first cell is composed of NAND cells while the rest are NOR. the pre-charge occurs only if a matching case happens in the first cell otherwise the pre-charge transistor of the first cell is discounted from the charge line. Hence, the power consumed by the search lines is saved [56,57].

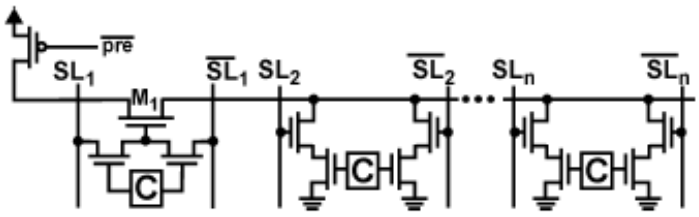


Figure 3.13 : Selective pre-charge matching.

3.4.5.3. Pipelining scheme

In the pipelining match lines scheme, the match line is split into any number of segments. in which the occurrence of a match in a segment leads to the continuation of the search process in the next segment. Otherwise, if a mismatch occurs the search operation is terminated. Figure 3.14 gives an example of a match line split up into four segments. where the serial evaluation of the match line segments is shown. If any of the

segments are mismatched, the following segment is turned off, resulting in energy savings by shutting down after missing a stage. The implementation of this shape has disadvantages of increased latency and the increase in the used area due to the pipeline segments [58].

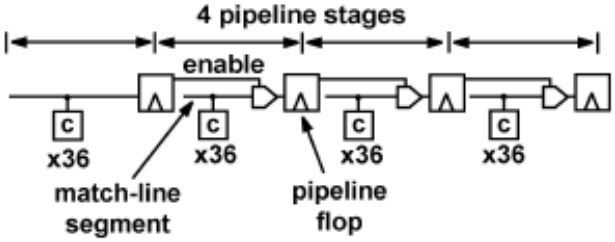


Figure 3.14 : 4 pipelined stages matchline sensing.

3.4.6. Searchline driving approaches:

3.4.6.1. Conventional approach

The traditional method of driving the search lines is applicable to match line schemes that pre-charge the match lines high. While searching, a cascade of inverters connects the search lines to their pre-charge level and then to their data value.

3.4.6.2. Eliminating searchline precharge

By eliminating the search line pre-charge stage, the search line power can be saved where the toggling of the search lines is reduced, and thus power is reduced.

3.4.6.3. Hierarchical searchlines

A hierarchical Search lines scheme is another method for reducing the power consumed by CAM's search lines where some search lines are turned off when it is possible.

3.4.7. CAM’s working principle:

CAMs are typically composed of M-word x N-bit cells, with input provided as a search word. These search word bits are streamed into corresponding search lines, where they are compared to a table of stored data (CAM bit cells). The size of the CAM is often

enormous in terms of bits, ranging from a few to a large number depending on the number of cells required for performing operations or the resources available on the board being used. The length of the words represents the width of the CAM (columns), and the number of words represents its depth (rows). The address space of the CAM is defined by the number of words used due to the uniqueness of each address, using a larger number of words results in a larger address space. For example, a 132X16 CAM would require 132 bits to represent its addresses, while a 512X8 CAM would require 512 bits. The CAM search operation begins by inserting the search-for word into the search-word register. All matchlines of the CAM are then pre-charged high, putting them in the match state. The search word is then loaded into the corresponding search lines through the search-line driver. A comparison operation is then performed between the stored CAM cell bit and the corresponding search line bit using logical operators. where XOR, NOR, and NAND gates are usually used to implement the bit comparison operations.

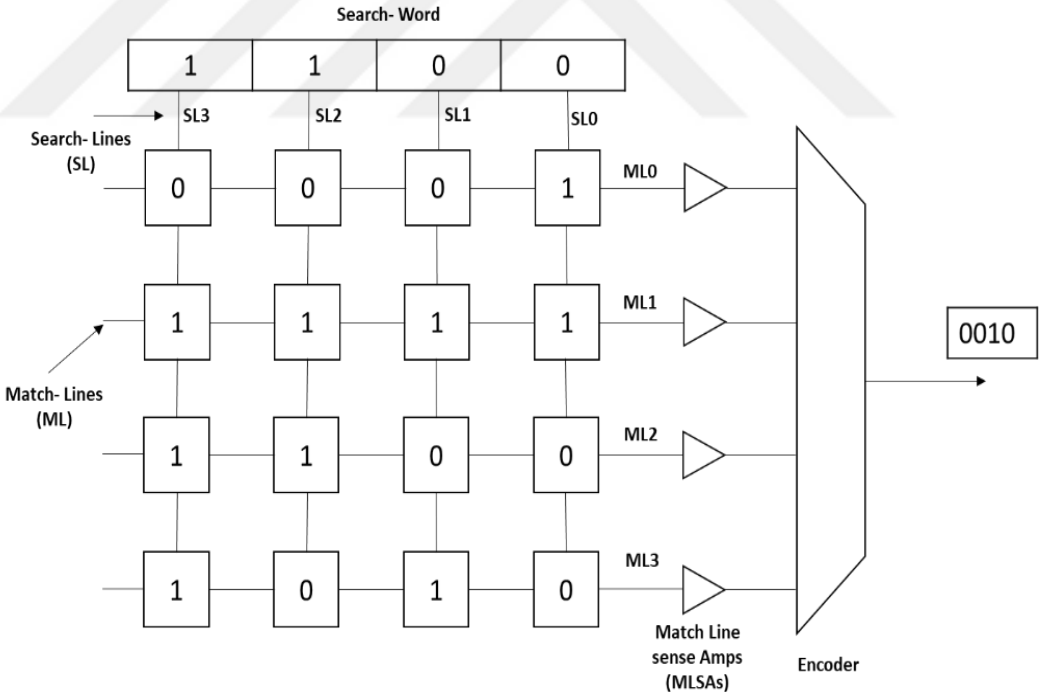


Figure 3.15 : An example of 4x4 CAM implementation.

Figure 3.15 is presented to give a clearer picture of how CAM works. We can notice that each word is made up of 4 bits arranged horizontally. The key register has the value of ("1100") that is loaded to search lines. where only the third word of the CAM

("1100") matches all the bits in the search line which will set the corresponding match line to high, while all the other match lines are grounded. Finally, the encoder gives the address of the matched word which is ("0010").

3.4.8. CAM's difficulties

The CAM's search operation speed comes at the expense of increased power consumption due to a large amount of parallel switching circuitry and a larger silicon area [53, 59]. Therefore, as CAMs become more popular and their applications grow, designers are attempting to reduce these two issues of size and power consumption. The main focus of recent research in large-capacity CAMs is to reduce power consumption without compromising area or speed. There are generally two basic methods for lowering the power dissipation of CAM cells: at the circuit design level and at the architectural level. Most efforts have focused on the architectural level due to the difficulty of improving the circuit or transistor-level design of CAM cells [35]. One method that significantly reduces power consumption is pipelining (selective pre-charge), which divides a matchline into two parts and only checks the remaining bits if a matching occurs in the first part. Another method for reducing power consumption is the hierarchical search line, which turns off search lines when possible.

3.4.9. Advantages and disadvantages of CAMs

The advantages and disadvantages of CAM are summarized in Table 3.2 below.

Table 3.3 : Advantages and disadvantages of CAM.

Advantages	Disadvantages
• Appropriate for parallel searche operations .	• It costs more than RAM because each cell must have additional storage capacity as well as logic circuits for matching.
• It is regularly utilized to accelerate databases, neural networks, and the tables used by modern computers' virtual memory.	• CAM suffers from power consumption associated with a large amount of parallel active circuitry
• Used when search time is critical and constrained	• Due to a large amount of parallel active circuitry, CAMs must have more space

CHAPTER 4. FPGA-BASED CONTENT ADDRESSABLE MEMORY (CAM)

4.1. Introduction

In the previous chapter, the designs of conventional CAMs were presented. where these kinds of CAMs are called Circuit-level or transistor-level CAMs, and they could be referred to as ASIC-based CAMs in literature. The Circuit-level CAM's cost shows a difference according to the used number of transistors per CAM cell. This happens due to the difference in hardware construction of both BiCAM and TCAM, and based on the type of logic gate, the CAM cells may be composed of either NAND-cell or NOR-cell. Where generally NAND-cell has 9 transistors while NOR-cell has 10 transistors [51,60]. The custom production feature of conventional CAMs (Circuit-level) makes them unusable in case of manufacturing error as well as unscalable. On the contrary, we find that FPGA -based CAMs are becoming more popular due to their FPGAs' hardware-like performance and software-like reconfigurability. Especially in complex reconfigurable systems. This could be a useful technology to implement the CAM as it can support highly parallel searching operations and can return the address of any given search word in a single clock cycle. Unfortunately, even though CAMs could be critical for some applications such as networking (routing), the current FPGA boards don't contain a hard IP CAM within their architecture [45]. However, modern FPGA boards offer an enormous number of logic and memory resources that can be used to design CAM and implement CAM-based applications.

Each FPGA device is made up of a specific number of resources, programmable interconnects, and I/O blocks that enable the implementation of reconfigurable digital circuits. A general FPGA structure is presented in Figure 4.1. The CLBs resources and routing resources are the two significant components of the structure which are arranged

in a matrix format. The CLBs resources, sometimes known as slices or logic cells are the basic logic unit of an FPGA and consist of two fundamental parts: Flip-flops (FFs) and lookup tables (LUTs). Where they are mainly utilized to implement logic functions in a digital design. The CLBs provided by Xilinx' are an example of CLBs, where each CLB is split into two slices. Each slice contains FFs, multiplexers, carry chain logic, and LUTs [61]. The routing resources connect several logic blocks for creating the required digital system. Generally, six transistors make up each switching node of the routes, and an SRAM configuration memory control them [45]. In Figure 4.2 a modern FPGA architecture is shown.

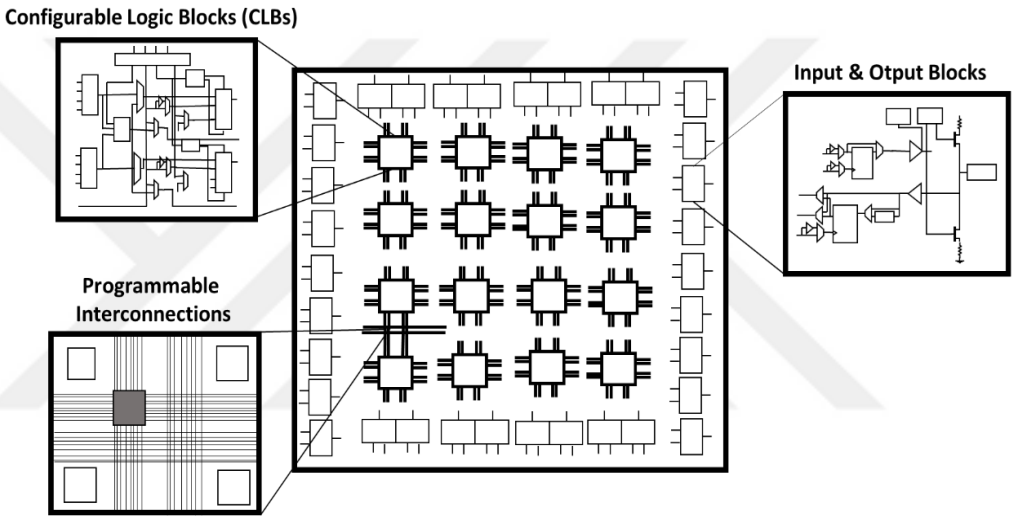


Figure 4.1 : The main components of the conventional FPGAs.

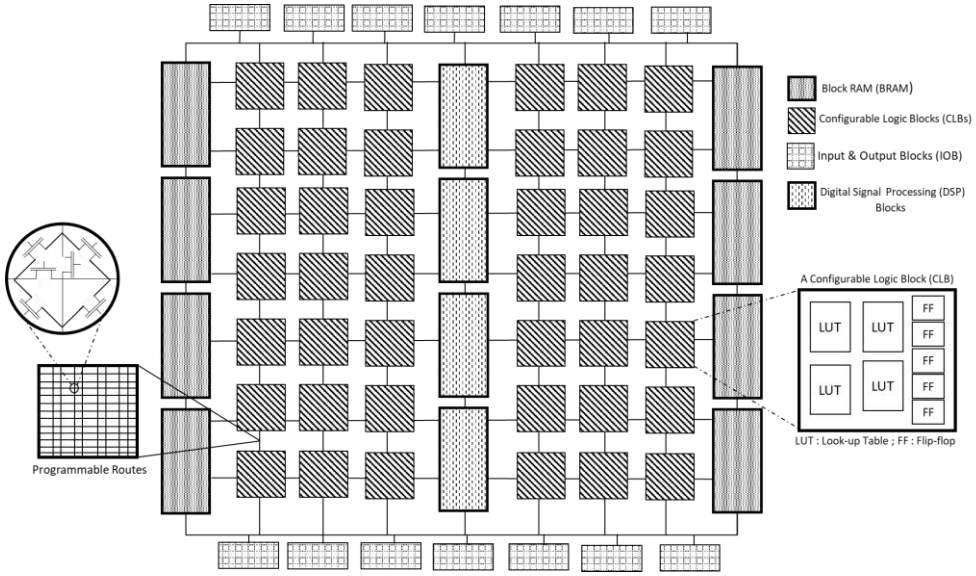


Figure 4.2 : A modern FPGA architecture.

4.2. Manufacturers of FPGAs' Boards

The main FPGA manufacturers are Xilinx, Altera, Microsemi, and Lattice Semiconductor [62]. In previous cores of Altera (now Intel) FPGAs FLEX and Mercury, CAM was supported in the form of Embedded System Blocks (ESBs), which could be configured as 32x32 BiCAM but it is no longer supported [63,64]. Xilinx does not provide a built-in hardware component that can be configured as CAM. However, software cores called intellectual property (IP) are released multiple times [65,75]. Where it is constructed using reconfigurable components (CLBs, RAM blocks, and other elements) to implement the CAMs.

4.3. Memories of Modern FPGAs

Understanding the nature of memories in FPGAs is essential as the FPGA-based CAMs are typically built-up using FPGA memories. Block RAM (BRAM), Look-up Table (LUT) RAM, and flip-flops are the main three types of memory that are presented in modern FPGAs.

4.3.1. Block RAM (BRAM)

Block Random Access Memory (BRAM) is a type of RAM used for data storage that is integrated into FPGAs. BRAMs are designed for quick access to large amounts of data, such as lookup tables and arrays of coefficients. They have a dual-port design that enables them to be accessed from two different sources simultaneously. Figure 4.3 shows the block diagram of the BRAM. Table 4.1 provides a summary of the implementation of BRAM-based CAMs in the literature.

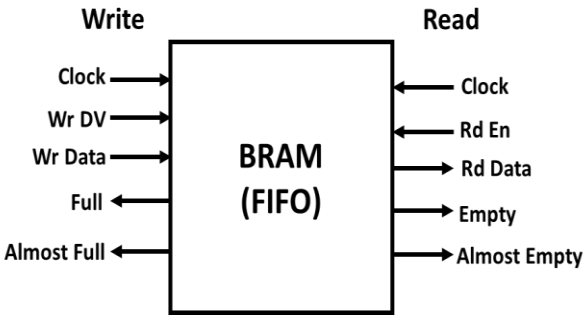


Figure 4.3 : BRAM Block diagram.

Table 4.1 : Summary of BRAM-based CAMs.

Architecture	Used Device	Size	Hardware Utilization	Characteristics
HP-TCAM [46]	Virtex-6	512 × 36	56 (36k) BRAMs	• First-time FPGA implementation (+). • Redesigning ont considered (-). • Data order must be ascending or descending (-). • Data preparation is necessary (-).
Z-TCAM [67] & E-TCAM [68]	Virtex-6	512 × 36		• Reduced search latency (+). • Redesigning not considered (-). • Data order must be ascending or descending (-). • Data preparation is necessary (-).
UE-TCAM [69]	Virtex-6	512 × 36	32 (36k) BRAMs	• Hardware efficiency increased (+). • Reduced search latency (+). • Data order must be ascending or descending (-). • Redesigning not considered (-).
EE-TCAM [70]	Virtex-6	512 × 36	32 (36) BRAMs	• Decreased power usage (+). • Increased search latency (-).
Scalable TCAM [71]	Virtex-7	1024 × 150	1024×150 (36) BRAMs	• There was a consideration of redesigning (-).
Hierarchical TCAM [72]	Virtex-6	504x180	140 (36) BRAMs	• Hardware efficiency increased (+). • Low latency (+).
Multi-Pumping TCAM [73]	Virtex-6	512x36	16 (36) BRAMs	• Hardware efficiency increased (+). • Low latency (+).

Architecture	Used Device	Size	Hardware Utilization	Characteristics
Xilinx TCAM (2011) (BiCAM) [74]	Virtex-5	1024× 36	104 Slices	• Hardware efficiency increased (+). • Low latency (+).
Xilinx TCAM for SDNet (2017) [75]	Virtex-7	512 x 40	3 (36K) BRAM 12.011 Slices	• The ternary CAM employing BRAMs is not supported (BiCAM) (-). • Low speed (-).

4.3.2. Look-up Table RAM (LUTRAM)

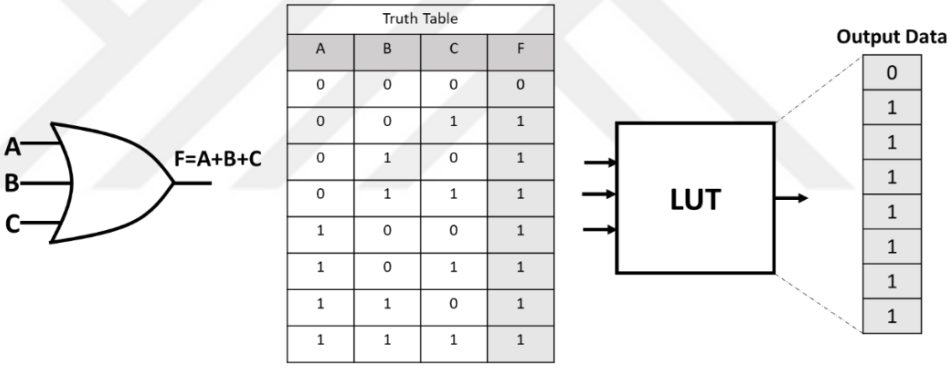


Figure 4.4 : A look-up table as a function generator.

Look-Up Table Random-Access Memory, or LUTRAM, is a sort of memory that combines the features of a LUT and a RAM. LUTs are a type of digital circuit that can be used to implement any Boolean function. Where they store a Boolean function in the form of a truth table. When the input values to the LUT are applied, the output value is determined by looking up the corresponding entry in the truth table. LUTRAMs are like regular RAMs. However, they may be set up to implement any Boolean function, making them more adaptable than conventional RAMs. This makes them useful for storing data such as lookup tables and arrays of coefficients that need to be accessed quickly. Figure 4.4 shows an example of a Look-up table with its truth table and logic gate symbol. Mostly, FPGA devices can utilize just one-third of their total LUTs as

LUTRAMs. whereas the remaining two-thirds can only be utilized to implement logic operations [45]. Both BRAM and LUTRAM use Sequential data access and it's limited to one access per time. Thus, to implement a search operation many accesses may be required. Table 4.2 gives a summary of implementing LUT RAM-based CAMs in the literature.

Table 4.2 : Distributed RAM-based CAMs LUTRAM-based.

Architecture	Used Device	Size	Hardware Utilization	Characteristics
Zi-CAM [76]	Virtex-7	512×36	12.011 Slices	• Low power for a particular set of data (+). • Very low speed (-). • Ternary bits are not supported (-).
Hierarchical TCAM [72]	Artix-7	504×180	894 Slices	• Low power for a particular set of data (+). • Low latency (+).
Xilinx CAM 2017 [77]	Virtex-6	1024×64	10.067 Slices	• Inefficient LUTRAM usage (-).
Scalable TCAM [71]	Virtex-6	4096×150	814 Slices	• A large Impleme-ntation size (+). • The update module was displayed (+).
D-TCAM [78]	Virtex-6	512×36	968 Slices	• High throughput (+). • Unable to update module (-).
PR-TCAM [79]	Virtex-7	512×40	20.526 Slices	• Addition of partial reconfiguration (+). • Slow update procedure (-).
DURE [80]	Virtex-6	512×36	1668 Slices	• Easily updated dynamism (+). • Low throughput (-).

4.3.3. Flip-flop (FF)

Flip-Flops (FFs) are one-bit memory components that can be used as a data storage component or pipeline registers in complex digital systems to increase system speed. It can store only two states either high "1" or low "0" and has a clock as input for synchronization. FFs are capable to be combined numerously with providing simultaneous access to the data contained in them. In Table 4.3 provides a summary of implementing Flip-flop (FF)-based CAMs in literature.

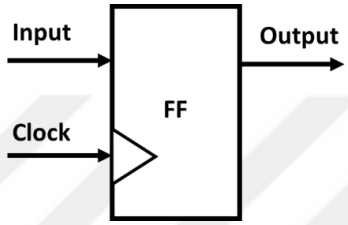


Figure 4.5 : FF's Block diagram.

Table 4.3 : Flip-flop (FF)-based CAMs.

Architecture	Used Device	Size	Hardware Utilization	Characteristics
RE-TCAM [81]	Virtex-6	64 × 36	629	• More I/O pins are needed (-). • Input data should be masked (-). • Utilized hardware resources are reduced (+);
LH-CAM [49]	Virtex-6	64 × 36	315	• Improved speed in comparison to BRAM-based CAMs (+). • Complex routing (-). • Unscalable (-).
G-AETCAM [82]	Virtex-6	64 × 36	774	• First FF-based TCAM (+). • Better performance compared to BRAM-based CAMs (+). • Unscalable (-).

By examining the existing developments and needs of CAM, we find that numerous publications have mentioned the necessity of implementing CAMs using soft-core techniques rather than commonly used ASIC-based techniques that use only hardware components to make them able to reconfigure and develop. To achieve that, many FPGA-based CAM optimizations are suggested in the literature as shown in this chapter. In comparison to BRAM-based and LUTRAM-based CAMs, FF-based CAMs are less scalable. But it reveals more effectiveness in terms of throughput (speed), energy consumption, and hardware resource utilization for small-size CAMs. Future CAM designs may reduce the number of FFs needed per CAM cell. This may increase its scalability, which is a fascinating topic for future studies.

CHAPTER 5. FPGA-BASED BiCAM IMPLEMENTATION AND EVALUATION

5.1. Introduction to The Chosen Methodology

In this thesis, a novel approach to enhance the speed of search operation within the memory in computing systems is proposed. This approach suggests a hardware-based enhancement by replacing the usually used storage unit structure (like RAM) with FPGA-based BiCAM. The proposed approach can improve the functionality of the system in terms of the speed of operations within the system. Multiple search algorithms have been introduced in the literature to improve the speed of the search operation, but most of them rely on software-based enhancements, not on hardware-based ones [83-87]. In this thesis, a novelty comes by substituting the RAM with the proposed FPGA-based BiCAM. In other words, to enhance the search operation speed a hardware-based modification is made rather than the commonly used software-based methods.

The BZK.SAU.FPGA microcomputer is used as a case study to demonstrate and evaluate the proposed approach. The Cyclone® II 2C70 core is chosen as the FPGA board to implement the proposed approach as it is the same board used to implement the BZK.SAU.FPGA [22]. The architecture of the proposed approach is entirely built at the logic gate level using Altera's Quartus II software. The programming language used to design the proposed approach is VHDL. Where it is one of the hardware description languages (HDL) that can model the structure and the behavior of the digital systems at multiple levels, from the entire system to the logic gate as designing entries. Which makes it more popular as a programming language for complex digital electronic circuits. The FF-based approach is chosen for implementing the BiCAM as it provides a more efficient design in terms of speed (throughput), power consumption, and hardware

resources. where in our case study there is no need to design a large-size BiCAM [45]. For the proposed approach design a D-type Flip-flop and a logic-based comparator circuit are used to implement the Bitcell of the BiCAM. While designing the proposed approach, the modularity of the design was taken into consideration, which is an important factor that allows for easier management of complex designs and facilitates future enhancements or adjustments.

To evaluate the effectiveness of the proposed approach, two different implementations were carried out. The first implementation used the proposed BiCAM, and the second implementation used a RAM-based brute-force search algorithm. The brute-force search algorithm, which is a linear search operation, was chosen as the method for the RAM-based implementation. This algorithm was used in the implementation of BZK.SAU. FPGA 10.1 as well [88].

The steps used to design the proposed approach can be briefly explained as follows: Firstly, a bit cell module is designed which consists of a flip-flop connected to a logic-based comparison circuit, which is in turn connected to the corresponding bit of the Key and Argument registers to perform the comparison operation. Next, the designed bit cell is replicated to form an n-bit column of bit cells and connected to the corresponding match line. After that, the n-bit columns (string) of bit cells are replicated to form an m-word (row), that will create an m-by-n matrix of bit cells, these will give the final appearance of the BiCAM main block. A decoder and encoder module are designed to be used in converting the bit size of the address and match line to make them more understandable for the user. A read-only memory (ROM) is used to initialize the values of the BiCAM, and a delay module is used for synchronization between the ROM and the BiCAM. Detailed information about the design of the proposed approach is provided in the following subsections.

5.2. Altera DE2-70 Board

Altera DE2-70 is an FPGA board used for educational and development purposes .it contains about 70,000 logic elements LEs with Altera Cyclone® II 2C70 core. It has numerous features and large memory components that enable the user to implement a wide variety of designed circuits ranging from simple logic circuits to various

multimedia projects [89]. Which makes it suitable for use in a laboratory environment for both universities and advanced digital systems development centers. The Altera DE2-70 board provides an ideal way to learn about digital logic, computer organization, and FPGAs' structure and implementation. It uses modern technologies for both hardware and software tools to introduce students and professionals to a wide range of circuit design-related topics. In figure 5.1, the overall view of the Altera DE2-70 board is given with an illustration of its components. In figure 5.2, the block diagram of the Altera DE2-70 board is given with mentioning each block's main features. Also, a features summary of Cyclone II 2C70 FPGA is given in Table 5.1. The Software tool that is provided to the DE2theard features is the Quartus® II Web Edition CAD software.

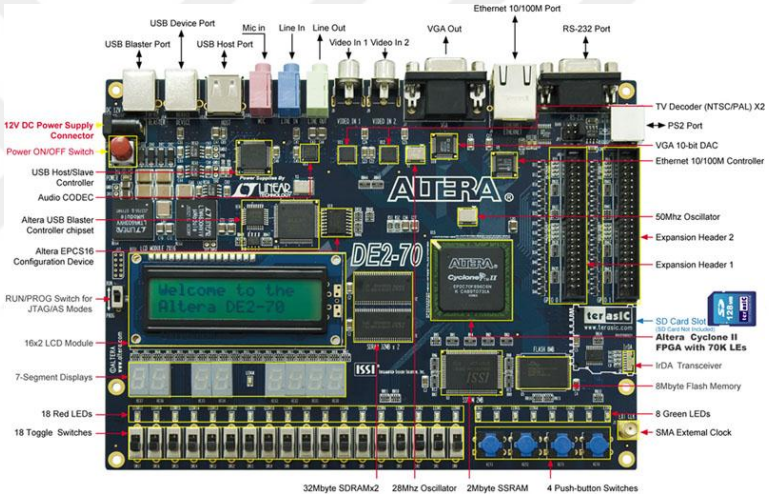


Figure 5.1 : The DE2-70 board.

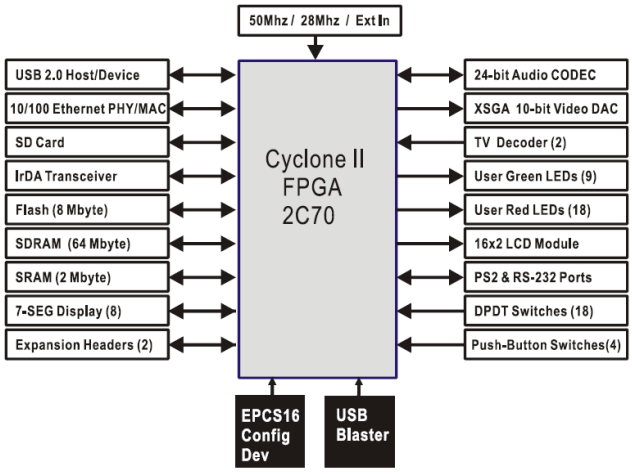


Figure 5.2 : The DE2-70 board block diagram.

Table 5.1 : Cyclone II 2C70 FPGA features summary.

Feature	Amount
LEs	68,416
M4K RAM block	250
Total RAM bits	1,152,000
Embedded multipliers	150
PLLs	4
User I/O pins	622
FineLine BGA	896-pin

5.3. Quartus II 13.0sp1 (64-bit) Web Edition

Altera Quartus II is software that is used as a platform design environment for both FPGAs and Complex programmable logic devices CPLDs. Altera Quartus II software can be easily adapted to the specific needs of different designs and provides a high level of productivity and performance. It can support both VHDL and Verilog implementations as HDL, visual editing of logic circuits (Block Diagrams), and vector waveform simulation. Altera Quartus II 13.0sp1 Web Edition is a free version of the Altera Quartus II software. It supports the HDL design for all process stages of logic circuits. including entries design and analysis, synthesis, placement, and routing. Also, it allows design compiling, performing timing analysis, investigating RTL diagrams, simulation, and configuring the target device with a programmer. This software can be considered a useful software tool for designing any type of logic circuit from simple and small learning circuits to large and complex designs that may be found in today's commercial products. The Quartus II software provides superiority in terms of synthesis, placement, and routing, resulting in faster compilation times [90]. The following features of Quartus II software contribute to speeding up compilation time:

- Multiple processors are supported.
- Fast recompilation.
- Incremental compilation.

In December 2015 Intel completed the acquisition of Altera and the Altera Quartus II software become named Intel Quartus II [91]. In Figures 5.3 & 5.4 the logos of both Altera and Intel Quartus II are shown.

Figure 5.3: Altera Quartus II's logo.

Figure 5.4 : Intel Quartus II's logo.

5.4. FPGA-based BiCAM

In general, we can say that the proposed FPGA-based BiCAM design consists of 5 main modules as follows: BiCAM, mem (ROM), delay, counter, and decoder modules. Each module of them will be explained in detail in this subsection. In figure 5.5 the general RTL diagram of the proposed approach design is shown. Also, the block diagram symbol was created and shown in Figure 5.6. From block diagram, we can see that the proposed design has 8 inputs and 3 outputs. The inputs are as follows: clock (CLK), mode-selection (mode_sel), write or read enable (W_R), reset, 8-bit Address, 64-bit Argument register (Argument_reg), 64-bit key register (Key_reg), and 64-bit data input (din). The outputs are as follows: 64-bit BiCAM output (BiCAM_out),16-bit machine code (machine_code), and 8-bit address for the match flag (match_falg_adrs).

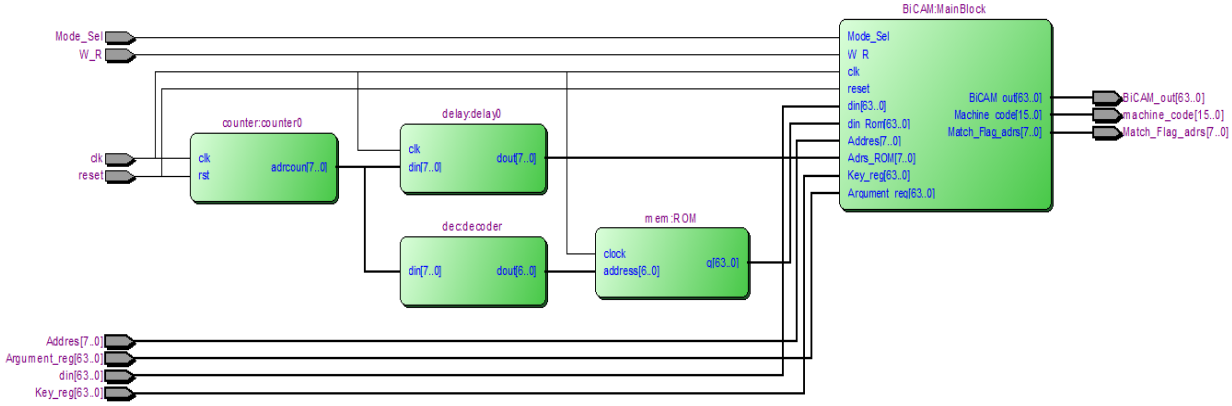


Figure 5.5 : The RTL diagram of the proposed FPGA-based BiCAM.

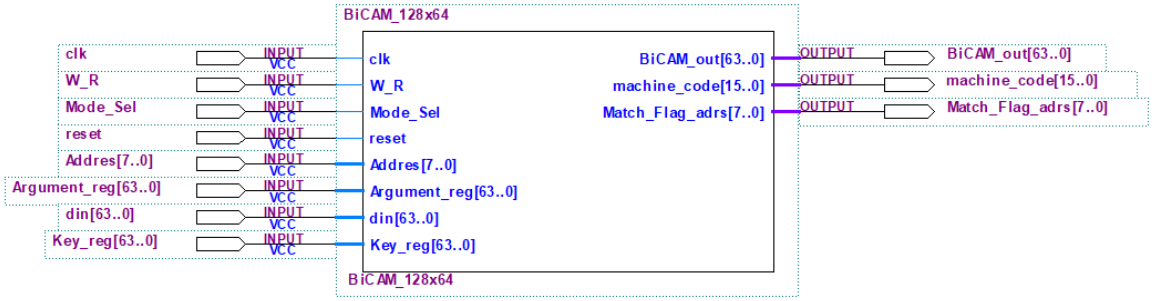


Figure 5.6 : Block diagram Symbol of the proposed FPGA-based BiCAM.

5.4.1. BiCAM : Mainblock

The BiCAM: Main block module is the essence of the proposed approach. It contains most of the main components of the design. This module consists of multiple submodules that will be explained in detail below. In figure 5.7 the RTL block diagram of the BiCAM Main block is given where th inputs and outputs of the module can be observed.

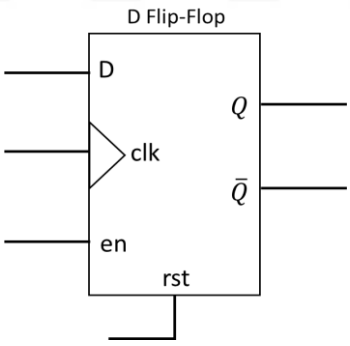
Figure 5.7 : Block diagram Symbol of the proposed FPGA-based BiCAM.

5.4.1.1. Bitcell

Each bit cell of the BiCAM consists of two main components. The first is a FF where data can be stored or retrieved. The second is a logic-based comparison circuit to determine if a match occurs between the stored value in the flip-flop and the search value that comes from the argument register. Each bit cell of the BiCAM is equipped with a logic-based comparison circuit that is connected to its corresponding flip-flop's output and the search value as inputs, where a logical comparison operation is performed. The output of this comparison circuit is connected to the match flag register

5.4.1.1.1. Flip-Flop (FF)

A Flip-flop is a circuit that can store a single bit of data in the state of "0" or "1" and retrieve it when required [92,93]. It is one of the essential components and it is considered as the memory element of the proposed BiCAM design. To implement this memory element, a simple D-type FF module is designed. The designed module has 4 inputs as follows: clock (Clk) that is used for synchronization, input data (data_in), a read/ write enable (w/r enable) that is used to determine if the FF is whether in reading or writing mode, and an address enable to activate the related FF cell. Both w/r and address enable are logically ANDed to enable the related FF to store the input data or retrieve it. Also, the FF module has two outputs: data output Q and its complement $\bar{Q}$. Briefly, it can be said that the designed FF module takes a single input data to be stored. Then, pass it toward the logic-based comparison circuit when needed. In figure 5.8 D-FF symbol and its truth table are given. The RTL diagram of the designed FF and its symbol of the block diagram is given as well in figure 5.9 and 5.10 respectively.



D Flip-Flop Truth Table

CLK	en	rst	D	Q	$\bar{Q}$
↓	0	0	0	Q	$\bar{Q}$
↓	0	0	1	Q	$\bar{Q}$
↓	1	0	0	0	1
↓	1	0	1	1	0
↓	1	1	0	1	0
↓	1	1	1	1	0
x	x	x	x	Q	$\bar{Q}$

X= Don't Care Case

Figure 5.8 : D Flip-flop Symbol and its Truth Table.

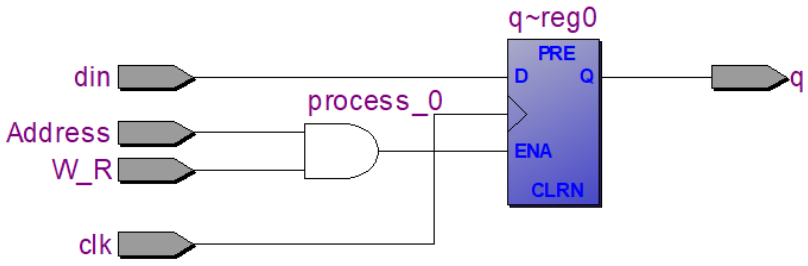


Figure 5.9 : The RTL diagram of the D flip-flop of the proposed BiCAM.

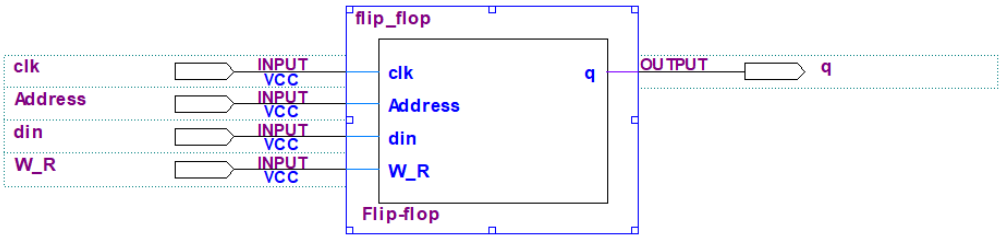


Figure 5.10 : The D Flip-flop block diagram of the proposed BiCAM.

5.4.1.1.2. Comparison circuit

The logic-based comparison circuit aims to determine if a match occurs between the search data and the stored data in the related FF. In case of a match, the output of the circuit is set to high "1", otherwise it is set to low "0". Each bit cell has its logic-based comparison circuit where it logically compares each data-bit individually. The comparison circuit has three inputs. They are the output of the FF Q, key_register, argument_register, and one output M. the output of the comparison circuit is connected to its related matchline. The comparison operation of the circuit is done logically by using OR_gate, AND_gate, and NOT_gate according to the equation 5.1 [94] that is given below:

$$\boldsymbol{M = A_{reg}F_{out} + \overline{A_{reg}}\,\overline{F_{out}} + \overline{K_{reg}}} \qquad (5.1)$$

Where M is equal to the output of the comparison circuit, F_{out} is equal to the value that comes from the output of the Flip_flop, A_{reg} is equal to the value that comes from the Argument register, K_{reg} is equal to the value that comes from the Key register. In figure 5.11 below the RTL diagram of the comparison circuit has been given:

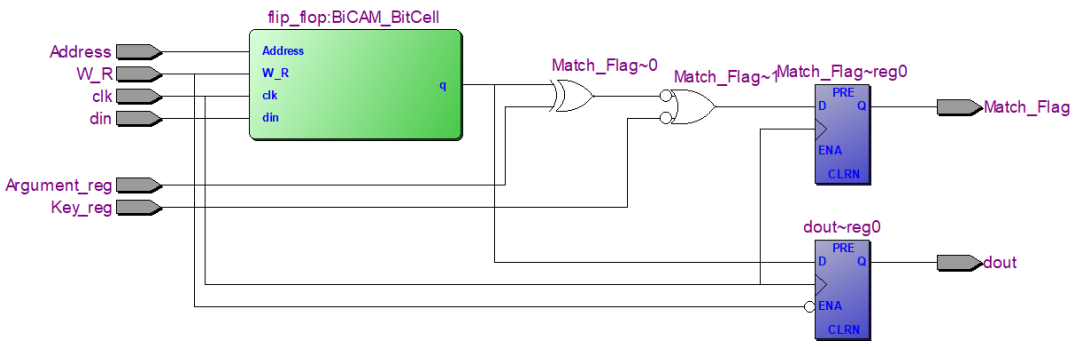


Figure 5.11 : The RTL diagram of the comparison circuit of the proposed BiCAM.

In addition, a mode select option is added to the terminal of the comparison circuit. It allows the user to choose whether he wants to activate or deactivate the comparison circuit. In other words, it gives the option to read the value stored in the desired FF directly without applying the comparison operation. If the comparison circuit is deactivated and the related address bit is high, the output of the FF is outputted without applying the comparison circuit. This selection can be achieved using a 2x1 multiplexer, as shown in Figure 5.12 below.

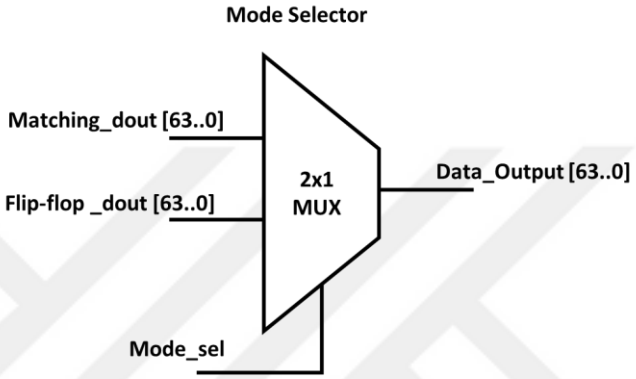


Figure 5.12 : The 2x1 Multiplexer that is used as a mode selector in the proposed BiCAM.

5.4.1.2. Row (String) of Bitcells

After completing the design of the bit cell module, the designed bit cell module is duplicated sequentially to form a row (string) consisting of 64 columns using the Generate function in the VHDL language. The Generate function is a function that is used to automatically repeat modules using the same signals, processes, and instances. The number 64 was chosen because the BZKSAU assembler command size may vary, but the BiCAM should have a fixed size and be unscalable after synthesis. Thus, the number 64 is chosen according to the size of the longest command that BZKSAU can have. Each row of the BiCAM has its own matchline that is linked to the outputs of the comparison circuits of each bit cell of the corresponding row. In Figure 5.13, a simplified visualization of the 64-bit row is given.

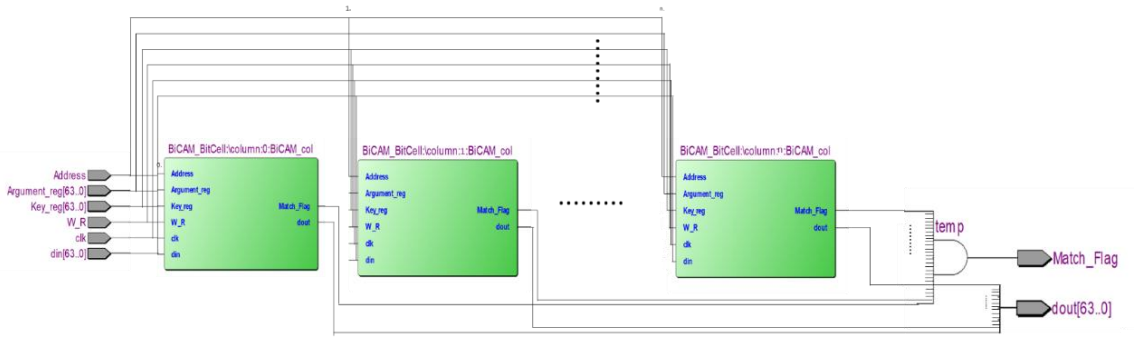


Figure 5.13 : The string (Row) architecture of bit cells of the proposed BiCAM.

5.4.1.3. Array (Matrix) of Bitcells

To achieve the array-shaped design of the proposed BiCAM, the Generate function is used again to create a 128x64-bit cell array (matrix), consisting of 64 columns and 128 rows. The number of rows, which is 128, was chosen based on the number of command sets that the BZKSAU assembler has. In Figure 5.14, a simplified visualization of the design of the 128x64 BiCAM is provided.

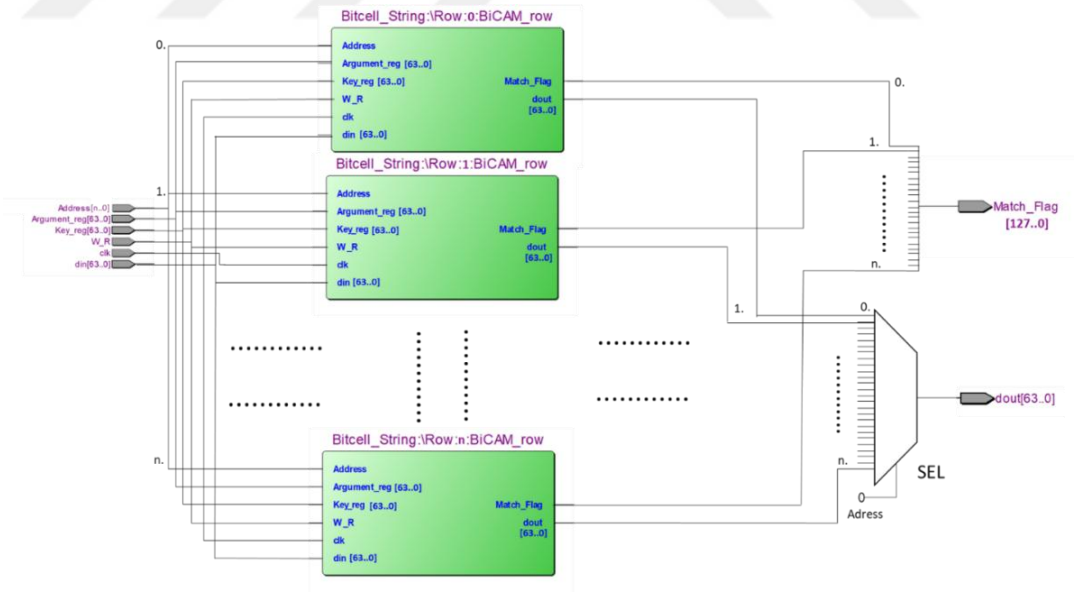


Figure 5.14 : The array (matrix) architecture of bit_cells of the proposed BiCAM.

5.4.1.4. Key_register

The key_register is a 64-bit string (array) that is connected bit-by-bit to the input of each comparison circuit of each bit cell. The key_register is one of the crucial inputs of the proposed BiCAM. It is used to implement a bitmasking operation on the argument register, filtering out unwanted bits of data and setting them to a don't care state. Figure 5.15 shows an illustrative example of the key_register.

Figure 5.15 : 64-bit Key_register.

5.4.1.5. Argument _register

The Argument_register is a 64-bit string (array) that serves as the primary input to the proposed BiCAM, where the desired search data (word) should be inserted to be searched for within the BiCAM. The Argument_register has 64 bits that are connected bit-by-bit to the input of each comparison circuit of each bit cell. Figure 5.16 shows an illustrative example of the Argument_register.

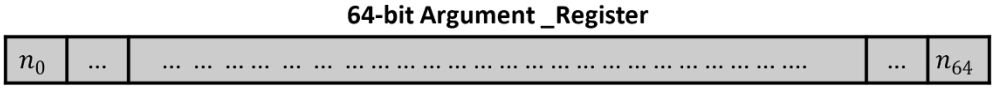


Figure 5.16 : 64-bit Argument_register.

5.4.1.6. Decoder

A decoder is a combinational circuit that can convert binary data from n input bits to 2^n output bits, or in other words, from one format to another format. Having a large number of address bits can be difficult for a user to assign and manage. For example, instead of assigning a 16-bit long address, a 4-bit address can be assigned and used to access the same desired BiCAM cell address using a decoder. In our case, as each row of the proposed BiCAM has its own unique address (the corresponding bit of the row is high and the remaining bits are low), it requires a relatively large number of address bits

that need to be minimized. That's why an 8 to 128 decoder is used, as shown in Figure 5.17.

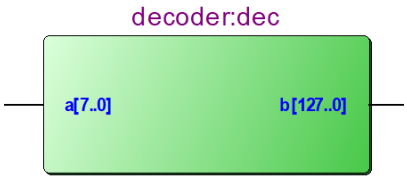


Figure 5.17 : The RTL diagram block of 8 to 128 decoder.

5.4.1.7. Encoder

An encoder is the reverse of a decoder, which is a combinational circuit that can convert binary data from 2^n input bits to n output bits. Each row of the proposed BiCAM represents a single data (word) and is connected to its match line, which is already connected to the 128-bit match flag register. The large bit size of the match flag register can be confusing and difficult for users to handle, so an encoder is used to simplify the address of the match flag register for the user. In our case, a 128 to 8 encoder is used, as shown in Figure 5.18.

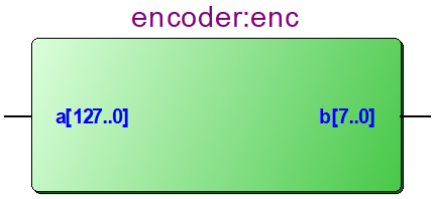


Figure 5.18 : The RTL diagram block for 128 to 8 encoder.

5.4.2. Single-port ROM :

To initialize the proposed BiCAM with the desired initial data, a storage unit is needed to load the initial data into the BiCAM during the initialization process. A read-only memory (ROM) is suggested as the storage unit, as it is suitable for the proposed design since only loading data into the BiCAM is required. Vendor-specific intellectual property core IP blocks are a useful tool and have become a common design methodology for larger and more complicated designs. Megafunctions are complex or high-level building IP blocks provided by Altera [95] that are parameterizable IP blocks

used to optimize Altera device architectures. Using Megafunctions instead of writing your own code can be a time-saving method when designing FPGAs. Altera provides multiple types of memories in Megafunctions such as ALTSYNCRAM, ALTDPRAM, ALT3PRAM, and LPM_RAM_ DQ Megafunctions [96]. The Quartus® II software automatically chose the type of memory according to the target device, memory modes, and features of the memory's functions. For our proposed design, the altsyncram is chosen as the DE2-70 board is used. MegaWizard plug-ins is a graphical user interface GUI used by Altera to enable the quick configuration of Megafunctions. It allows creating or modifying the design of custom Megafunctions. MegaWizard can be found in the tools list of Quartus II. MegaWizard provides the lpm_rom Megafunction as a ROM in two different modes 1-port ROM and 2-port ROM which can be found in the memory compiler file. The 1-port lpm_ROM Megafunction is chosen for the proposed design. where it is fully parameterizable and allows both synchronous and asynchronous single ports. The lpm_ROM needed to be initialized with desired initial data while configuring. A memory initialization file (.mif) or hexadecimal (.hex) file should be used with the same name as the lpm_ROM in the project directory. In figure 5.19 the interface of the MegaWizard Plug-In Manager where the adjustment of the lpm_ROM parameters such as the required number of bits and words are made is given.

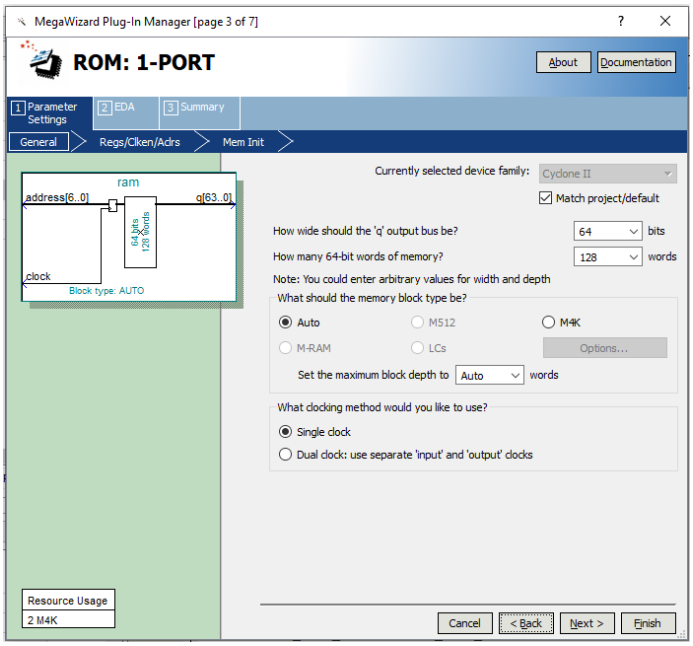
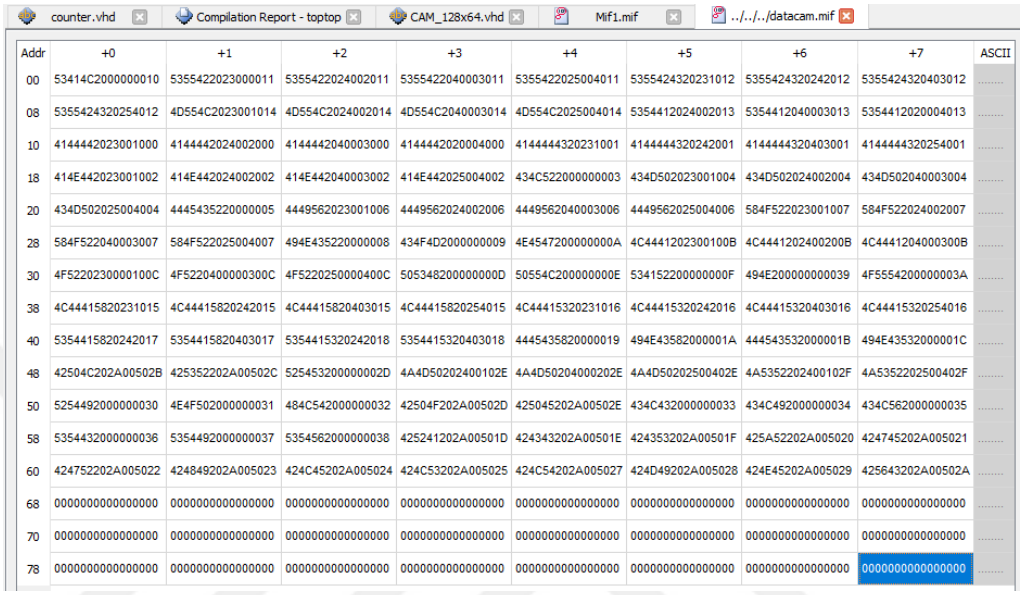


Figure 5.19 : Interface of lp-ROM in MegaWizard Plug-In.

The memory initialization file (.mif) or hexadecimal (.hex) that is used for the ROM initialization can be created using The Quartus II software. in figure 5.20 the memory initialization file (.mif) that is used for initialization ROM is shown. The output file is shown in figure 5.21.



counter.vhd | Compilation Report - toptop | CAM_128x64.vhd | Mif1.mif | ../../../datacam.mif

Addr	+0	+1	+2	+3	+4	+5	+6	+7	ASCII
00	53414C2000000010	5355422023000011	5355422024002011	5355422040003011	5355422025004011	5355424320231012	5355424320242012	5355424320403012	.....
08	5355424320254012	4D554C2023001014	4D554C2024002014	4D554C2040003014	4D554C2025004014	5354412024002013	5354412040003013	5354412020004013	.....
10	4144442023001000	4144442024002000	4144442040003000	4144442020004000	4144444320231001	4144444320242001	4144444320403001	4144444320254001	.....
18	414E442023001002	414E442024002002	414E442040003002	414E442025004002	434C522000000003	434D502023001004	434D502024002004	434D502040003004	.....
20	434D502025004004	4445435220000005	4449562023001006	4449562024002006	4449562040003006	4449562025004006	584F522023001007	584F522024002007	.....
28	584F522040003007	584F522025004007	494E435220000008	434F4D2000000009	4E4547200000000A	4C4441202300100B	4C4441202400200B	4C4441204000300B	.....
30	4F5220230000100C	4F522040000300C	4F522025000400C	505348200000000D	50554C200000000E	534152200000000F	494E200000000039	4F5554200000003A	.....
38	4C44415820231015	4C44415820242015	4C44415820403015	4C44415820254015	4C44415320231016	4C44415320242016	4C44415320403016	4C44415320254016	.....
40	5354415820242017	5354415820403017	5354415320242018	5354415320403018	4445435820000019	494E43582000001A	4445435320000001B	494E43532000001C	.....
48	42504C202A00502B	425352202A00502C	525453200000002D	4A4D50202400102E	4A4D50204000202E	4A4D50202500402E	4A5352202400102F	4A5352202500402F	.....
50	5254492000000030	4E4F502000000031	484C542000000032	42504F202A00502D	425045202A00502E	434C432000000033	434C492000000034	434C562000000035	.....
58	5354432000000036	5354492000000037	5354562000000038	425241202A00501D	424343202A00501E	424353202A00501F	425A52202A005020	424745202A005021	.....
60	424752202A005022	424849202A005023	424C45202A005024	424C53202A005025	424C54202A005027	424D49202A005028	424E45202A005029	425643202A00502A	.....
68	0000000000000000	0000000000000000	0000000000000000	0000000000000000	0000000000000000	0000000000000000	0000000000000000	0000000000000000	.....
70	0000000000000000	0000000000000000	0000000000000000	0000000000000000	0000000000000000	0000000000000000	0000000000000000	0000000000000000	.....
78	0000000000000000	0000000000000000	0000000000000000	0000000000000000	0000000000000000	0000000000000000	0000000000000000	0000000000000000	.....

Figure 5.20 : Quartus II memory initialization File.

```
-- Quartus II generated Memory Initialization File (.mif)

WIDTH=64;
DEPTH=128;

ADDRESS_RADIX=HEX;
DATA_RADIX=HEX;


CONTENT BEGIN
        00  :   53414C2000000010;
        01  :   5355422023000011;
        02  :   5355422024002011;
        03  :   5355422040003011;
        04  :   5355422025004011;
        05  :   5355424320231012;
        06  :   5355424320242012;
        07  :   5355424320403012;
        08  :   5355424320254012;
        09  :   4D554C2023001014;
```

Figure 5.21 : The output .mif file.

5.4.3. Counter module

A counter module is designed to be used in the address incrementing process of both the lpm_ROM and the proposed BiCAM during initialization. The initial values of the

BiCAM are stored in the lpm_ROM. By incrementing the address of the lpm_ROM, the value stored at that address is taken and stored in the corresponding address of the proposed BiCAM. The module has two inputs: a clock for synchronization and a reset to reset the counter, and one output, which is the desired address. Figure 5.22 shows the RTL design of the module.

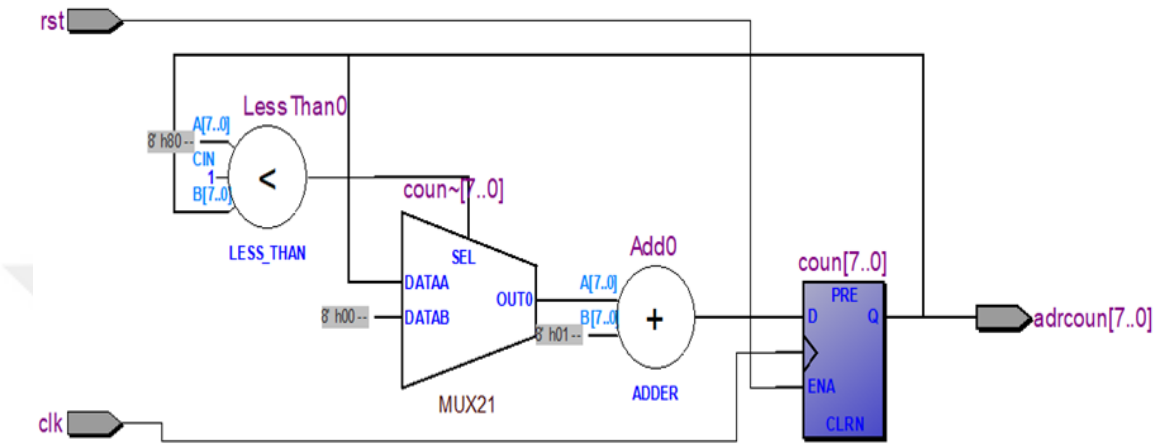


Figure 5.22 : The RTL diagram of counter module.

5.4.4. Delay module

A delay module is created to achieve synchronization between the lpm_ROM and BiCAM during the initialization process. This synchronization is achieved by applying a delay of one clock cycle, as the lpm_ROM output is delayed by one clock in the beginning. The delay module has two inputs and one output. The inputs are the clock and the address that come from the counter module. The output of the delay module is the same as its address input, but with a delay of one clock cycle. Figure 5.23 shows the RTL design of the delay module.

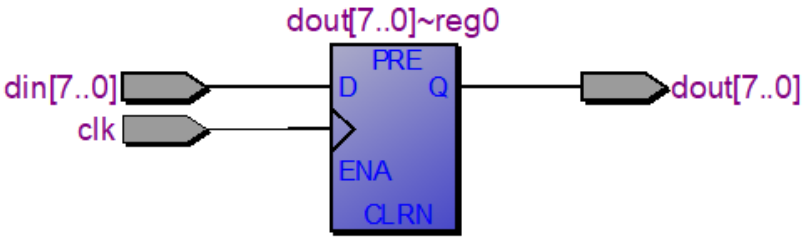


Figure 5.23 : The RTL diagram of delay module.

5.4.5. Counter Decoder for ROM module

Another decoder module is used to reduce the bit size of the address input that will be used in the already defined lpm_ROM Megafunction. The address input for this module is automatically set as 6-bit and cannot be changed manually. As the address input coming from the counter module has 7-bit, a decoder is needed to ensure compatibility.

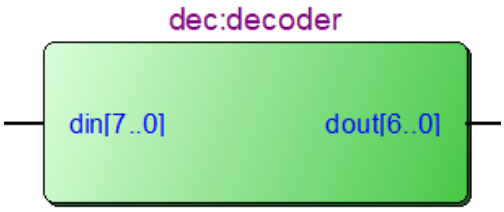


Figure 5.24 : The RTL module of the counter decoder for ROM.

5.5. Evaluation

The evaluation of the proposed BiCAM is conducted by carrying out two different FPGA-based implementations using BZK.SAU assembler. The first implementation replaces the RAM with the proposed BiCAM. The second implementation uses a RAM-based Brute-force algorithm to be compared with the proposed BiCAM. The BZKSAU assembly language statements can be divided into assembler directive (Opcode) and operand, and a search operation within the Opcode must be performed to find the corresponding machine language instruction to be executed by a processor or controller. The Opcode is a table consisting of a set of machine language instructions with their related mnemonics, address modes, and operands if needed. By using this table, the machine code of the operations to be executed by a processor or controller can be determined. The operand may contain data, addresses, or values if it is required by the used architecture. Using an FPGA-based BiCAM to optimize the search operation of machine language instructions within the Opcode naturally leads to an enhancement in the overall performance of the system.

To demonstrate the efficiency of the proposed approach, a comparison with another commonly used approach should be carried out. The Brute-force algorithm, which was used as the search algorithm in the BZKSAU assembler [21], is the chosen algorithm to perform the comparison with the proposed BiCAM. The Brute-force search algorithm

can be considered a type of linear (sequential) search operation, which can be executed simply by comparing the values sequentially until the desired value is matched [79]. To make the comparison more realistic and equal, the Brute-force search algorithm is implemented using VHDL as the programming language in Quartus II software and loaded onto a DE2-70 FPGA board (the same programming language and board used for designing the BiCAM). Both the BiCAM and the Brute-force algorithm approaches are implemented according to the BZK.SAU assembler architecture's criteria, meaning that they are both built as 16-bit systems and use the same Opcode table. A randomly picked group of assembly codes is used for the implementation.

Time complexity is the analysis used to measure the efficiency of a proposed approach. Comprehension of the time complexities of search algorithms can help determine which is more efficient and faster. Generally, the time complexity of search algorithms is measured using Big O notation. More details will be provided in the Results and Discussion chapter.

5.5.1. Implementation by using FPGA-based Brute-Force Algorithm

5.5.1.1. Data pre-preparation and mapping

A bunch of random assembly codes is selected and converted to their ASCII equivalent in hexadecimal notation, as the ASCII format was used for the assembly language of the BZKSAUFPGA assembler. Then, each of the converted codes is mapped into a 16-bit RAM, as the BZKSAUFPGA has a 16-bit architecture. Each ASCII character is composed of two hexadecimal (hex) characters, and each hexadecimal character is composed of 4 bits. Therefore, each row of the 16-bit RAM will contain two ASCII characters. For opcode mapping, the same opcode that was used in the BZKSAUFPGA is selected and mapped into a 16-bit RAM in the same manner as the random assembly codes were mapped. Figure 5.25 shows an example of a 16-bit RAM that contains both an opcode and a random assembly code to demonstrate the mapping operation.

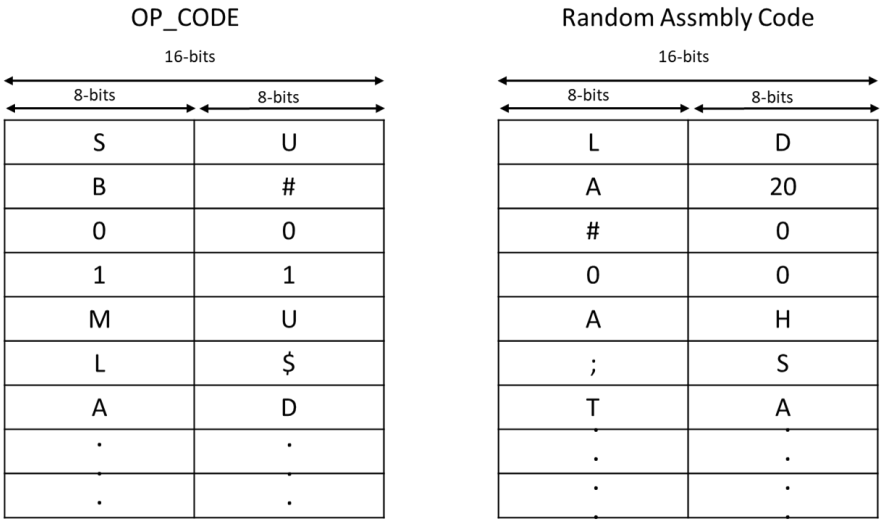


Figure 5.25 : Explanation of mapping operation of Opcode and random assembly code in RAM.

5.5.1.2. Brute-force algorithm-based search operation

After completing the data mapping process for both the opcode and the random assembly code, the search operation is implemented using the Brute-force algorithm, which uses the equality operator to perform the logic comparison operation to determine if both values are equal or not. To implement this, each row of the 16-bit RAM of both the opcode and the random assembly code is split into two equal portions (each has 8 bits), with the left portion being labeled as the most significant bits (MSB) and the right portion being labeled as the least significant bits (LSB), as shown in Figure 5.26.

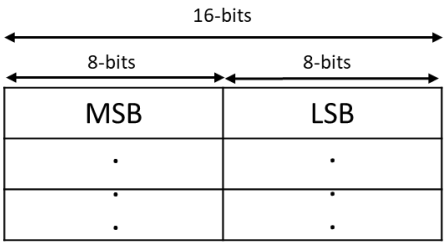


Figure 5.26 : The 16-bit RAM is split into 2 equal parts.

After that, the search operation begins by picking up the MSB part of the first row of the random assembly code and comparing it logically with the MSB and LSB parts of the first row of the Opcode respectively. The same search operation is continued for all rows of the Opcode until a match is found as shown in Figure 5.27.

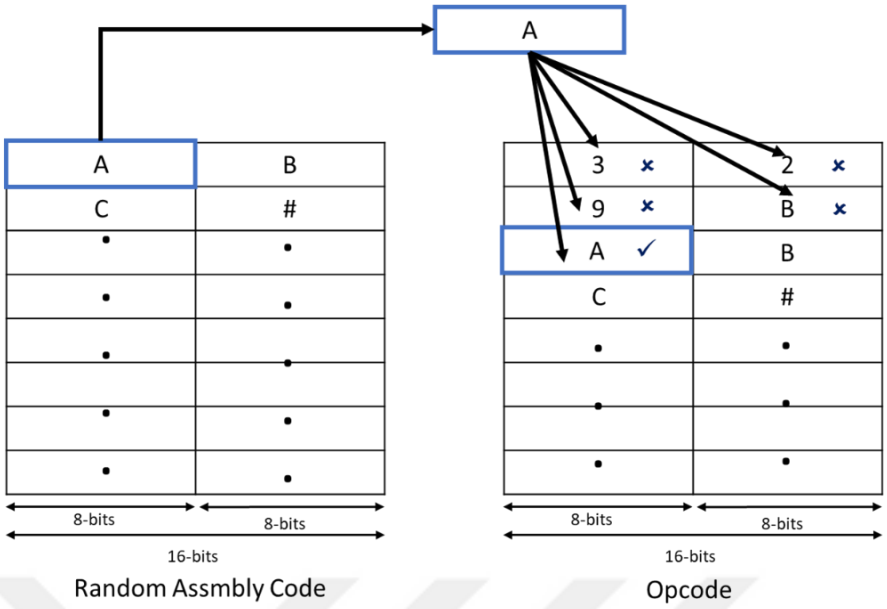


Figure 5.27 : Comparison Operation Using Brute-force Algorithm.

In case of a match occurs in the MSB or LSB part of the random assembly code, the next part to be picked for the continuation of the search operation is the LSB part if a match occurred in the MSB part. Otherwise, the next row's MSB part is picked if a match occurred in the LSB part. Then, the search operation continues from the same point where the previous match occurred in the Opcode. Figure 5.28 shows an example of these conditions.

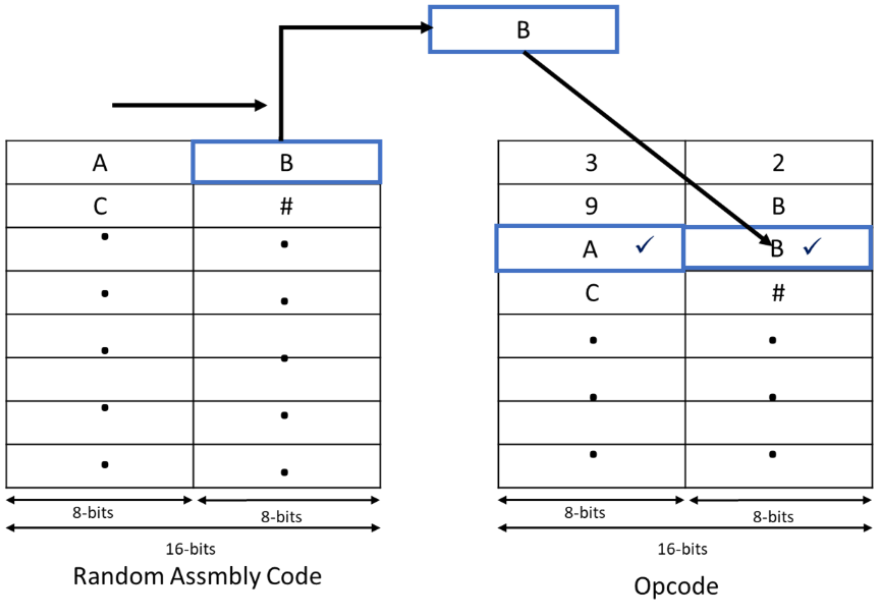


Figure 5.28 : Match condition of comparison operation using Brute-force algorithm.

As shown in Figures 5.29 & 5.30, if a mismatch occurs after a match or series of matches, the search operation restarts again from the first part (character) of the assembly command being searched. A pointer to the first character of each assembly command being searched is assigned during the search operation, which serves as a reference point to restart the comparison operation. The search operation continues like this until the end of the random assembly code.

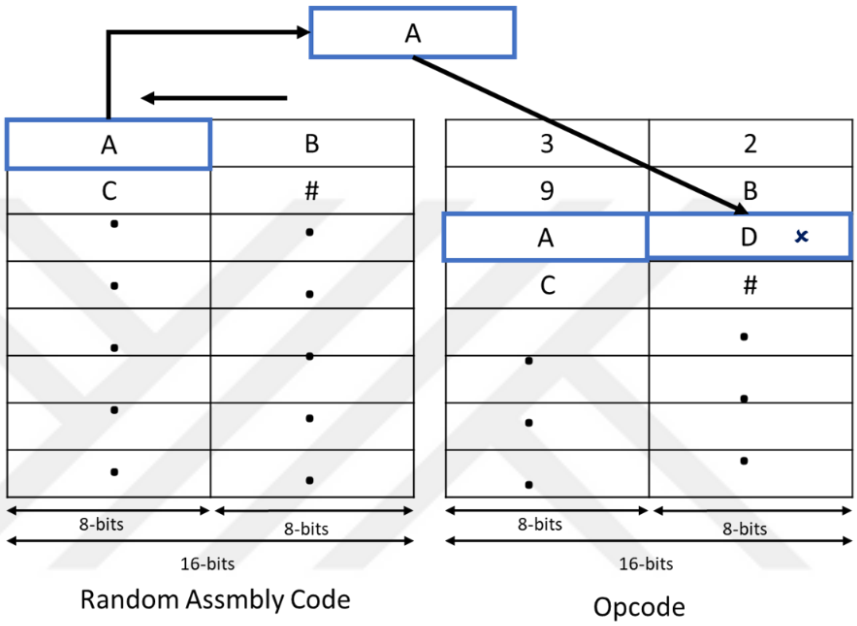


Figure 5.29 : Mismatch condition in the comparison operation using Brute-force algorithm.

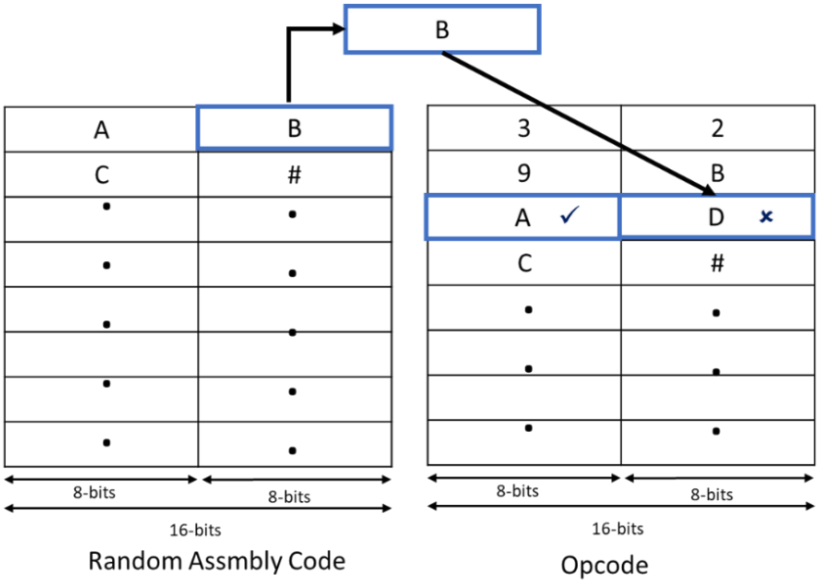
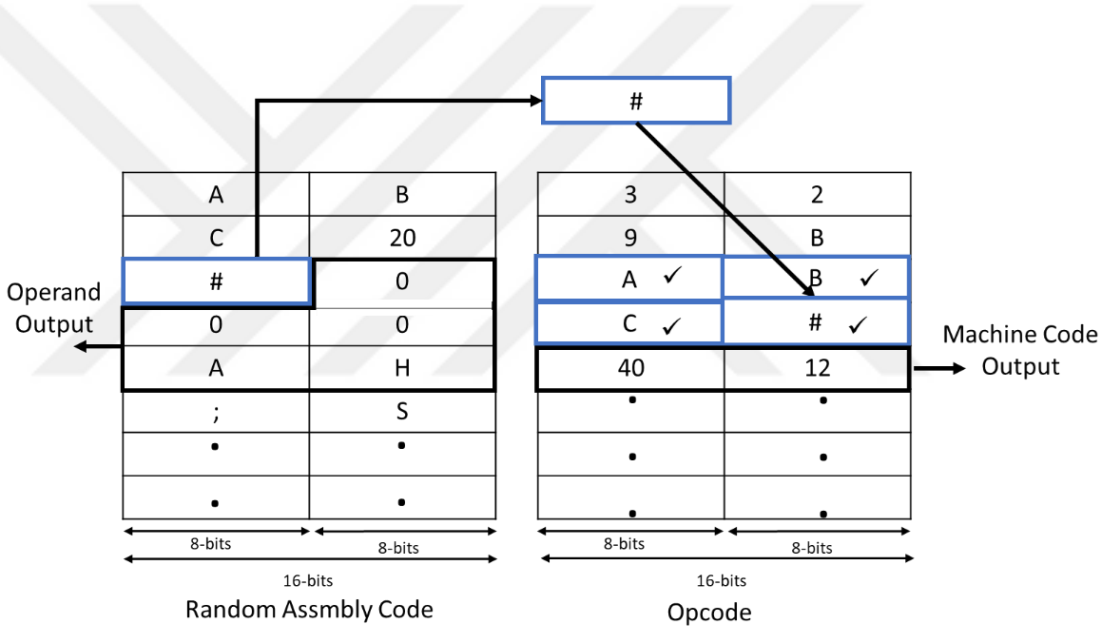


Figure 5.30 : Restarting search operation after mismatch.

The search operation continues until the desired sequence of both 16-bit RAMs is matched. The space character in the random assembly code is skipped during the search operation, as the Opcode does not include the space character. The end of the desired sequences is distinguished by the match of one of the addressing mode's characters. After that, the next 16-bit of the Opcode is extracted as output, which contains the required machine code. At the same time, another output that contains the operand code (if it exists) is extracted from the random assembly code. The operand code can be distinguished by detecting the address mode character as the starting point and a semicolon character as the endpoint. An example of this condition is shown in Figure 5.31 for clearer demonstration.



* 20: ASCII code for space character in hexadecimal notation

Figure 5.31 : Desired sequence of the search operation.

5.5.2. Implementation by using FPGA-based BiCAM

5.5.2.1. Data pre-preparation and mapping

To implement the search operation using the FPGA-based BiCAM, there is no need to map the Opcode table, as it is already mapped into a 16-bit ROM that will be used for the initialization process, as explained previously. For random assembler codes, the

designed 16-bit RAM in the implementation of the Brute-force algorithm is used, but it cannot be used directly. Instead, it must be arranged in the standardized form of the argument register. As the argument register is composed of 64 bits, the mapping operation begins by taking the most significant bits (MSBs) of the random assembly command and replacing them with the relevant MSBs of the argument register until the address mode character is detected, which indicates the end of the desired portion of the assembly command. The remaining bits of the argument register are set to the "don't care" condition, as they will be neglected in the later search operation. The operand of the assembly command is extracted in the same way as the argument register. An example of this is given in Figure 5.32.

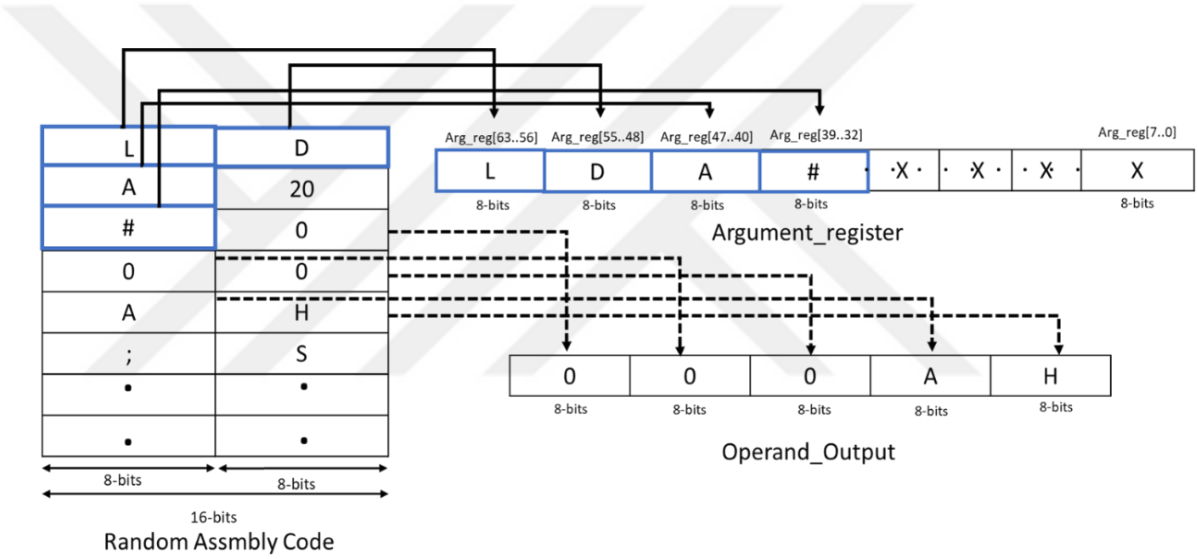
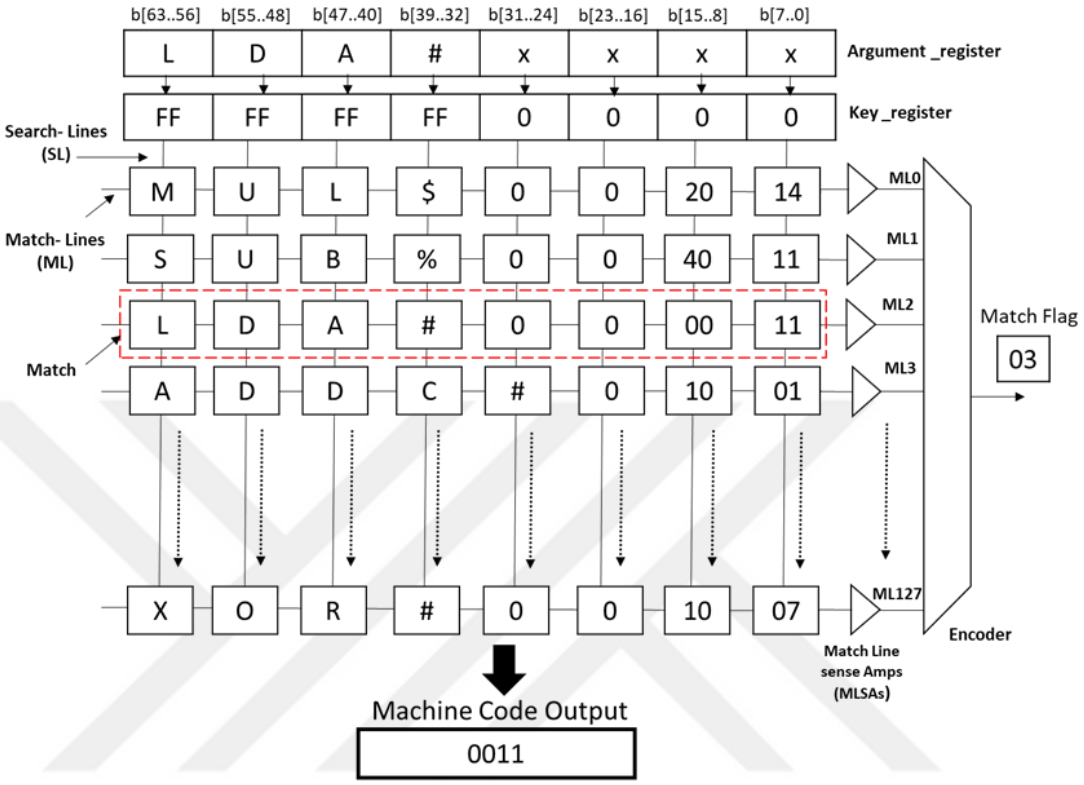


Figure 5.32 : Data preparation of Argument_register.

5.5.2.2. BiCAM-based search operation

The search operation is performed in parallel according to the proposed BiCAM's working principle. The search operation begins by loading the desired (search) data into the argument register. Then, a bitmasking operation is applied to the argument register using the key register, where the bits that are considered for the search operation are set to high '1' and the rest of the bits are set to low '0'. After that, the proposed BiCAM implements its search operation in parallel according to Equation 1. In a match case, the match flag of the corresponding row is set to high '1', and the relevant machine code bits are extracted as output. In whcih , the machine codes are located at the last 16 bits

(LSB) of the BiCAM's row. An example of a BiCAM-based search operation is shown in Figure 5.33 for demonstration.



* x = don't care state.

Figure 5.33 : An example of a BiCAM-based search operation.

CHAPTER 6. RESULTS AND DISCUSSIONS

6.1. Searching Algorithms

A search operation is the process of thoroughly seeking a target within a location. For example, if a list of data is given, the operation of comparing all the data with the desired data to find its location in the list is called a search operation. Search algorithms are designed to improve the efficiency and accuracy of search operations, leading to an improvement in system performance. Common search algorithms are split into three categories depending on their searching technique [98], as follows:

- **Linear search** (Sequential), which simply checks the values in sequence until the desired value is found.
- **Binary search**, which requires a sorted input list, and checks for the value in the middle of the list, discarding half of the list which contains values that are either larger or smaller than the desired value.
- **Hashing-based** search uses a hash table to store the data within, the data is mapped according to its key which is generated by using a hash function.

The hashing-based search algorithm is a software-based search algorithm and is considered the fastest search algorithm. It uses a table in array format called hash table to store the data associatively [99]. Where each data value of the hash table has its own index that is generated by using a hash function. The hash function converts one or more properties of the data to a value that can be used as an index into a hash table. This makes the search operation and accessing the desired stored data value faster. Figure 6.1 shows an explanation of the hash function process.

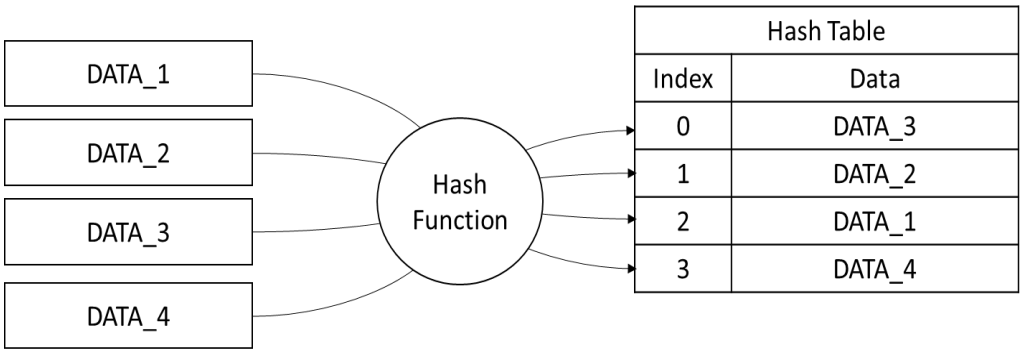


Figure 6.1 : Explanation of the hash function process.

In some cases, especially when the hash table is small, a collision occurs when the hash function gives the same index value to more than one data. This collision can be resolved by using different hashing techniques. One of the techniques is called linear probing. Which, stores the data in the next empty index to the calculated index. To do that, a linear search operation need to find the nearest empty index. Also, to find these stored data another linear search operation should be involved. The more data that is stored in the hash table the more the data indexes get collision. Another way to adress this , is to use a bigger hash table. Hashing is used widely in database indexing, compilers, caching passwords, authentication, and more .

6.2. Time Complexity

Analyzing algorithms is an essential aspect of computer science. It helps to determine which one of the algorithms is the most effective for a given problem. although, different algorithms can be used to solve the same problem, determining which one is the most efficient is still a challenge. It is commonly known that , an algorithm provides a satisfactory solution when it generates the correct answer and the amount of time required to execute a given algorithm can be used to estimate the algorithm's efficiency. Computers can perform the same operations by using different algorithms, but a suitable algorithm must be built to perform operations effectively in computers. That’s why algorithm analysis should be done to determine which one of the algorithms is more efficient than the others.

Even though there are various ways to compare algorithms, Time complexity, which is an essential analysis for algorithms, is being used within the scope of this thesis. Time

complexity investigates the number of steps required for algorithm implementation based on its input size [100] . There are three major factors that must be considered when comparing algorithms in the time complexity domain. These factors are:

I. The Execution Time: although it is considered a major factor. but due to the possibility of getting manipulated by computers. It cannot be the only factor used to compare algorithms.
II. The number of commands to execute: this is an important factor that is needed for the comparison. but the count of the commands to be executed may vary depending on the used algorithm and the used programming language. It also can't be the only factor used to compare algorithms.
III. The Ideal solution: it is the factor of representing an algorithm by its ideal runtime as a function of input size (n) independent of its programming style, computer time, and so on.

In time complexity analysis, the growth rate over time with respect to the size of the inputs while algorithm execution is calculated. Where it considers how many times each statement get executed rather than the real-time needed to execute each statement in the code. Time complexity can be measured with the help of Asymptotic notations of asymptoticanalysis. Asymptotic notations are mathematical notations used to describe an algorithm's runtime when its input size varies [101]. The primary goal of the asymptotic analysis is to provide a measure of the efficiency of algorithms without relying on computer parameters and does not require another algorithm to implement, and the time required by the whole program to be compared. The three important asymptotic notations of Time complexity are [101]:

- Big-O Notation (O): This notation represents the upper bound of an algorithm's run time, indicating whether functions grow faster than or at the same rate as the given expression.
- Omega Notation (Ω): This notation represents the lower bound of an algorithm's run time, indicating whether functions grow slower than or at the same rate as the given expression.

- Theta Notation (Θ): This notation represents the upper and lower bound of an algorithm's run time (encloses the function from above and below), indicating whether functions grow lie between O and Ω faster than.

Understanding the time complexities regarding the searching algorithms helps to choose the best searching algorithm and determine which one is faster. As the search operation is generally implemented by applying logic-based comparisons between the data until the desired data is found, the number of comparisons required is determined by the total length of the storage unit, and whether the desired data is near the beginning or near the end of the storage unit. The time complexity of the search operation is commonly expressed as a function of n where n represents the number of elements that the algorithm searches within. where the Time complexity function estimates the number of operations required to run the algorithm depending on n elements. As a result, the increase of the search operation number as n increases can be predicted. The Time complexity function differ depending on the search algorithms. For example , The Time complexity of RAM-based Brute-force algorithm, which is used to be implemented in order to be compared with the proposed BiCAM approach , performs the comparison linearly depending on the length n of the storage unit. Where each element of the volume is compared to the search term requested. In the time complexity analysis for search algorithms, three different scenarios are possible. The scenarios are best-case, average-case, and worst-case scenarios . The best-case scenario occurs when the first data to be compared is the desired data, meaning that finding the desired data only requires a single comparison operation. On the other hand, the worst-case scenario occurs when the desired data is the last item in the storage unit, requiring the search operation to be applied to the entire storage unit to be found. In the average case, all possible inputs are considered, and the comparison time for all inputs is calculated. The calculated values are then added up and divided by the total number of inputs. The best-case, expected, and worst-case scenarios for the main searching algorithms are summarized in Table 6.1 .

Table 6.1 : Summary of case scenarios for main RAM -based search algorithms.

Algorithm	Best case	Average case	Worst case
Linear search	O (1)	O (n)	O(n)
Binary search	O (1)	O (log n)	O(log n)
Hashing-based search	O (1)	O (1)	O(n)

When the time complexity equals O(1), it indicates that the number of needed search operations is constant regardless of the input size (n). This is the best-case scenario for search algorithms. From Table 6.1 above, we can see that the best search algorithm is the Hashing-based search algorithm, which has a constant result of O(1) for both the best and average cases. Additionally, for the binary search algorithm, we can see that the average and worst case values depend on the logarithmic value of the length of the storage unit (n) rather than a constant value.

As expected, after implementing the proposed BiCAM, the Time complexity of finding the desired search data in all cases was found to be a constant value of O(1). This means that it only takes a single clock cycle to find a match, if it exists. This is due to the parallel nature of the BiCAM, which allows the search operation to be implemented in parallel and find the desired search data in a single operation, regardless of the size of the input or the BiCAM.

To have a more comprehensive understanding of the proposed approach, a source utilization and power efficiency analysis is conducted in addition to the time complexity analysis.

6.3. Source Utilization

A summary of the source utilization is provided in Table 6.2 below. As expected, the BiCAM approach utilizes more logical elements, with an increase of approximately four times. Whereas , the proposed approach consists of FF as the storage unit and uses decoders and encoders in its design . Additionally, we can see a significant difference between the total memory bits used in the two methods. In the RAM-based method,

factory-specified onboard memory bits were used, while in the BiCAM-based approach, FFs served as the storage element.

Table 6.2 : Source utilization of both RAM and BiCAM-based methods.

Memory Type	RAM-based	BiCAM -based
Total logic elements	230	835
Total combinational functions	206	777
Total registers	89	222
Total pins	57	398
Total memory bits	21,056	1,200

6.4. Power Efficiency

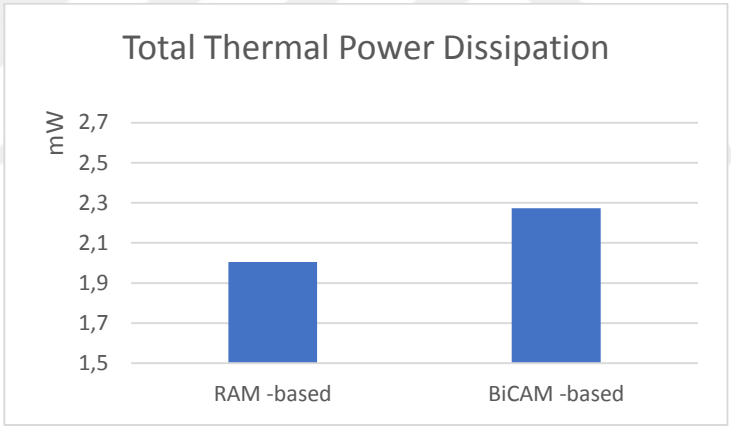


Figure 6.2 : Total thermal power dissipation graph of both BiCAM and RAM .

In Figure 6.2, a graph that shows us the total thermal power dissipation by both RAM and BiCAM-based methods is given . From the graph, we can notice that the BiCAM approach consumes more power than the RAM-based approach .The increase in power consumption is about 13.5%, due to the use of more logical elements. Despite this, the BiCAM-based approach is still suitable for time-sensitive applications, as previously mentioned.

CHAPTER 7. CONCLUSIONS

Memories are an essential part of computing systems and can affect the execution time of the systems, as they contain the instructions to be executed by the system's CPU or microcontroller. A search operation is required to find the desired instruction within a memory. Improving the search operation to make it faster is still necessary, although there are various techniques are suggested to do that, these techniques are often software-based and may have limitations due to the hardware structure of the memory used. In this thesis, a novel hardware-based enhancement to the search operation using the BiCAM structure as memory, rather than the existing software-based techniques, is proposed.

As a case study for this thesis, the BZK.SAU.FPGA microcomputer was used to implement the proposed BiCAM, which was also designed using the FPGA environment. Then, it was replaced instead of the RAM used in the BZK.SAU assembler, where the mnemonic instructions of the assembler were stored. The BZK.SAU is an assembler designed for educational purposes and used at Sakarya University since 2009. The BZK.SAU has its custom assembly language where each of its statements is divided into an opcode and an operand.

The proposed BiCAM was designed and implemented in an FPGA environment using an FF-based technique. From the designed BiCAM, it can be observed that its size and cost grow linearly with the length of data and the number of entries, due to its hardware structuring nature. However, it can still be considered a good trade-off as its enhancement in speed may be needed in time-sensitive applications. The proposed design may suffer from scalability issues, which are important in other applications where larger CAMs are required. However, it can be used in specific applications where no large-sized CAM is needed.

To verify the efficiency of the proposed approach, a RAM-based Brute-force algorithm was implemented in addition to the BiCAM implementation. The brute-force algorithm is a commonly used software-based algorithm for search operations and is considered one of the linear search techniques. Its implementation was done in the same FPGA environment for a fair comparison. As expected, the search operation using the Brute-force algorithm takes longer time (clock cycles) compared to the proposed BiCAM. In the Brute-force algorithm, the search key is compared to the entire memory row by row until a match occurs, while the BiCAM only requires a single clock cycle to find the match as it performs the comparison in parallel (all rows at the same time). The time complexity of the brute-force algorithm is generally O(n), where n represents the length of the search value. In the best case, the time complexity becomes a constant value O(1), meaning that the match occurs in the first searched value. However, in the BiCAM, the Time complexity is always O(1) regardless of the length of the search value or the number of entries. In addition to time complexity analysis, both power efficiency and source utilization analyses were conducted to provide a more comprehensive picture of the proposed approach. where the proposed BiCAM uses about 4 times more logical elements than the RAM-based brute force and consumes more power due to its hardware-based structure.

Another comparison to the proposed approach is done using software-based techniques, specifically the hashing-based algorithm, which is also a commonly used search operation and considered the best search operation technique as its time complexity value is constant in both the best and average cases O(1) and equal to O(n) only in the worst case. However, when compared to the proposed approach, the proposed approach maintains an advantage over it as its time complexity remains constant at O(1) in all cases. The hashing-based search algorithm may take more clock cycles to execute its hash function in order to generate the index where the data should be stored. Additionally, time may be wasted in the event of a collision, where a linear search operation may be required to find an empty index to store the data or to find the data if it is stored in another index due to the collision. Therefore, the proposed approach can be considered a hardware solution to overcome collisions in the hashing-based search algorithm.

To sum up, we can conclude that the proposed BiCAM shows better efficiency in terms of time complexity (speed), as the time taken to implement the search operation is equal to a constant value O(1) regardless of the length or size of the entries. Despite the speed advantage of the proposed approach, it comes with a drawback of higher cost compared to software-based techniques, as it is a hardware-based enhancement rather than a software one. However, it can still be considered a good trade-off in time-sensitive applications that are not too large in size.

REFERENCES

[1] Streib, J. T. (2020). *Guide to Assembly Language A Concise Introduction*. Cham, Switzerland: Springer.

[2] Austerlitz, H. (2003). *Data Acquisition Techniques Using PCs*. California, USA: Academic Press.

[3] Manjula, A. S. (2022). *System Software: Assemblers, Loaders And Linkers, Macro Processors, Compilers And Utilities*. Chhattisgarh, India: Orange Books.

[4] Salomon, D. (1993). *Assemblers and loaders*. West Sussex, UK: Ellis Horwood.

[5] Paul, J. F. & Howard E. M. (2003). Computer Systems Performance Evaluation and Prediction. USA: Digital Press.

[6] Page, A. & Waters, D. (2016). *Complete Computer Science for Cambridge IGCSERG and O Level Revision Guide*. Oxford, UK : Oxford University Press.

[7] Udipi, A. N., Muralimanohar, N., Chatterjee, N., Balasubramonian, R., Davis, A., & Jouppi, N. P. (2010). Rethinking DRAM design and organization for energy-constrained multi-cores., *Proceedings of the 37th Annual International Symposium on Computer Architecture - ISCA '10*. (pp- 175-186) Saint-Malo, France : June 19–23.

[8] Jian, X., Hanumolu, P. K. & Kumar, R. (2017). Understanding and Optimizing Power Consumption in Memory Networks. *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA),* (pp.229-240). Austin, TX, USA : February 4-8.

[9] Acar, H., Alptekin, G. I., Gelas , J. -P. & Ghodous, P., (2016). Beyond CPU: Considering memory power consumption of software. *2016 5th International Conference on Smart Cities and Green ICT Systems (SMARTGREENS),* (pp.1-8), Rome, Italy: April 23-25.

[10] Li, H. & Chen, Y. (2009). An overview of non-volatile memory technology and the implication for tools and architectures. *2009 Design , Automation & Test in Europe Conference & Exhibition.* (pp. 731-736). Nice , France: April 20-24.

[11] Amol, M. J. & Ajit, S. M. (2021). Data Structures using C: A Practical Approach for Beginners.London , UK : CRC Press.

[12] Fan, X., Ghonem, A., & Gemmeke, T. (2018). Content-Addressable Memory - Overview and Outlook of an Enabler for Modern Day Applications. *ANALOG 2018; 16th GMM/ITG-Symposium,* (pp.1-6). Munich/Neubiberg, Germany: September 13-14.

[13] Pagiamtzis , K., & Sheikholeslami , A. (2006). Content-addressable memory (CAM) circuits and architectures: a tutorial and survey , *IEEE Journal of Solid-State Circuits*, 41 (3),712-727.

[14] Hopfield, J. (1982). Brain, Computer, and Memory. *Engineering and Science*, 46 (1). 2-7.

[15] Lin, C .S., Chang, J. C., & Liu, B.D. (2003). A low-power precomputation- based fully parallel content-addressable memory, *IEEE Journal of Solid-State Circuits*, 38 (4), 654–662.

[16] Boutros, A. & Betz, V. (2021). FPGA Architecture: Principles and Progression, *IEEE Circuits and Systems Magazine*, 21 (2), 4-29.

[17] Trimberger, S. (1995). FPGA Technology: Past, Present, and Future. *ESSCIRC '95: Twenty-first European Solid-State Circuits Conference*, (pp. 12-15). Lille, France: September 19-21.

[18] Kuon, I., Tessier, R., & Rose,J. (2008). *FPGA Architecture: Survey and Challenges.* Massachusetts, USA :Now Publishers Inc.

[19] Kasivinayagam, G., Skanda, R., Burli, A.G., Jadon, S.,& Sidhu, R. (2022). Hardware Description Language Enhancements for High Level Synthesis of Hardware Accelerators. *Advances in Computing and Data Sciences*, 1613,1-12.

[20] Gandhare, S. & Karthikeyan, B. (2019). Survey on FPGA Architecture and Recent Applications. *2019 International Conference on Vision Towards Emerging Trends in Communication and Networking (ViTECoN*), (pp. 1-4). Vellore, India: March 30-31.

[21] Öztekin, H., Temurtas, F. & Gulbag, A.(2010). BZK.SAU.FPGA10.0: Microprocessor architecture design on reconfigurable hardware as an educational tool. *2011 IEEE Symposium on Computers & Informatics*, (pp. 385-389). Kuala Lumpur, Malaysia: March 20-23.

[22] Öztekin, H., Gülbağ, A. & Temurtaş , F. (2017). Assembler Design for BZK.SAU. FPGA Micro Computer Architecture. *Electronic Letters on Science and Engineering*, 13(1), 1-9.

[23] Öztekin, H., Temurtas, F. & Gulbag, A. (2010). BZK.SAU: Implementing a hardware and software-based Computer Architecture simulator for educational purpose. *2010 International Conference On Computer Design and Applications,* (pp.90-97). Qinhuangdao, China: June 25-27 .

[24] Öztekin, H., Temurtas, F. & Gulbag, A. (2014). BZK.SAU.FPGA10.1: A modular approach to FPGA-based micro computer architecture design for educational purposes. *Computer Applications in Engineering Education*, 22(2), 272–282.

[25] Öztekin, H. (2012). *Embedded Operating System Design on Configurable Modular Hardware for Educational Purposes*, (PhD Thesis), Sakarya University. Institute of Science and Technology, Sakarya.

[26] Temurtas, F. & Gulbag, A. (2012). *Educational Microcomputer Architecture and Embedded Operating System Design on Remote Accessible Configurable Hardware.* (Proj. No. 110E069). Yozgat : TÜBITAK-EEEAG.

[27] Url-1 < *http://en.wikipedia.org/wiki/ABC_80*>, date of access 24/11/2022.

[28] Mohammad, K., Qaroush, A., Washha, M., & Mohammad, B. (2017). Low-power content addressable memory (CAM) array for mobile devices. *Microelectronics Journal*, 67, 10-18.

[29] Sivakumar , S. A., Swedha, A., & Naveen , R. (2018). Survey of Content Addressable Memory. *International Journal of Creative Research Thoughts (IJCRT)*, 6 (1), 1516-1526 .

[30] Cheng, K. -H., Wei, C. -H. & Chen, Y. -W. (2003). Design of low-power content-addressable memory cell. *46th Midwest Symposium on Circuits and Systems*, (pp. 1447-1450). Cairo, Egypt : December 27-30.

[31] Hussain, S.W., Mahendra, T.V. , Mishra, S. ,& Dandapat, A. (2020). Low-Power Content Addressable Memory Design using Two-Layer P-N Match-Line Control and Sensing. *Integration*, 75 (),73-84.

[32] Chang, Y., Tsai, K., Cheng, Y., & Lu, M. (2020). Low-power ternary content-addressable memory design based on a voltage self-controlled fin field-effect transistor segment. *Computers and Electrical Engineering*, 81(C), 106528.

[33] Gangadhar, A., Sirisha, R. R., & Babulu, K. (2015). Implementation of content addressable memory with high speed and low power consumption. *2015 Conference on Power, Control, Communication and Computational Technologies for Sustainable Growth (PCCCTSG)*, (pp. 230-233). Kurnool, India: December 11-12.

[34] Satyanarayana, S. V. V. & Sridevi, S. (2019). Design of TCAM Architecture for Low Power and High Performance Applications. *Gazi University Journal of Science*, 32 (1) , 164-173.

[35] Jiang, S., Yan , P. & Sridhar ,R. (2015).A high speed and low power content-addressable memory(CAM) using pipelined scheme. *28th IEEE International System-on-Chip Conference (SOCC)*, (pp. 345-349). Beijing, China: September 08-11.

[36] Gamache, B., Pfeffer, Z., & Khatri, S.P. (2003). A fast ternary CAM design for IP networking applications. Proceedings. *12th International Conference on Computer Communications and Networks* (IEEE Cat. No.03EX712), (pp. 434-439), Dallas, TX, USA: 22-22 October.

[37] Ammendola, R., Biagioni, A., Frezza, O., Geurts, W., Goossens, G., Cicero, F.L., Lonardo, A., Paolucci, P.S., Rossetti, D., Simula, F., Tosoratto, L., & Vicini, P. (2015). ASIP acceleration for virtual-to-physical address translation on RDMA-enabled FPGA-based network interfaces. *Future Generation Computer Systems*, 53, 109-118.

[38] Yang, H., Wang, X., Yang, C., Cong, X. & Zhang, Y. (2019). Securing contentcentric networks with content-based encryption. *Journal of Network and Computer Applications*, 128 , 21–32.

[39] Imani , M., Peroni, D., Kim, Y., Rahimi, A. & Rosing, T. (2017). Efficient neural network acceleration on GPGPU using content addressable memory. *Proceedings of the Conference on Design, Automation & Test in Europe*,

European Design and Automation Association, (pp. 1026–1031). Lausanne, Switzerland: March 27-31.

[40] Le, D. H., Tran, B. T., Inoue, K. & Pham, C. K. (2014). A CAM-based information detection hardware system for fast image matching on FPGA. *IEICE Transactions on Electronics.* E97(C), 65–76.

[41] Mujahid, O. M., Ullah, Z., Mahmood, H., & Hafeez, A. (2018). Fast Pattern Recognition Through an LBP Driven CAM on FPGA. *IEEE Access*, 6, 39525-39531.

[42] Liu, A. X., Meiners , C. R. & Torng , E. (2016). Packet Classification Using Binary Content Addressable Memory. *IEEE/ACM Transactions on Networking*. 24(3), 1295-1307.

[43] Jothi, D. & Sivakumar, R. (2018). Design and Analysis of Power Efficient Binary Content Addressable Memory (PEBCAM) Core Cells. *Circuits, Systems and Signal Processing*, 37(6), 1422–1451.

[44] Zackriya , M. V. & Kittur , H. M. (2016). Precharge-Free, Low-Power Content-Addressable Memory. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems,* 24 (8), 2614-2621.

[45] Irfan, M., Sanka , Ullah , A.I., Z. & Cheung R.C.C. (2021). Reconfigurable content-addressable memory (CAM) on FPGAs: A tutorial and survey. *Future Generation Computer Systems,* 128, 451-465.

[46] Ullah, Z., Ilgon, K., & Baeg, S. (2012). Hybrid Partitioned SRAM-Based Ternary Content Addressable Memory. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 59, 2969-2979.

[47] Ahmed, A., Park, K., & Baeg, S. (2017). Resource-Efficient SRAM-Based Ternary Content Addressable Memory. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems,* 25, 1583-1587.

[48] Nash, J.G. (2018). Distributed-Memory-Based FFT Architecture and FPGA Implementations. *Electronics*, 7(7), 116–145.

[49] Ullah, Z. (2017). LH-CAM: Logic-Based Higher Performance Binary CAM Architecture on FPGA. *IEEE Embedded Systems Letters*, 9 (2), 29–32.

[50] Harris, M. H. & Harris, S. L. (2010). *Digital Design and Computer Architecture*, Massachusetts, USA : Morgan Kaufmann.

[51] Riazi, M.S., Samragh, M., & Koushanfar, F. (2017). CAMsure: Secure Content-Addressable Memory for Approximate Search. *ACM Transactions on Embedded Computing Systems,* 16(5S), 136 :1-20.

[52] Kumar, A., & Satyanarayana, B.V. (2018). Low voltage high speed 8T SRAM cell for ultra-low power applications. *International journal of engineering and technology*, 7(3.29), 70-74.

[53] Sattı, V. V. S., & Srıadıbhatla, S. (2019). Hybrid self-controlled precharge-free CAM design for low power and high performance. *Turkish Journal of Electrical Engineering and Computer Science*, 27 (2) , 1132–1146.

[54] Arsovski, I., & Sheikholeslami, A. (2003). A current-saving match-line sensing scheme for content-addressable memories, *IEEE International Solid-State Circuits Conference, 2003. Digest of Technical Papers. ISSCC*. (pp. 304-494). San Francisco, CA, USA : February 13.

[55] Mishra, S., Mahendra, T. V., Hussain, S. W., & Dandapat, A. (2020). The analogy of matchline sensing techniques for content addressable memory (CAM). *IET Comput. Digit. Tech.*, 14, 87-96.

[56] Zukowski, C. A., & Wang, S. (1997). Use of selective precharge for low-power content-addressable memories. *Proceedings of 1997 IEEE International Symposium on Circuits and Systems. Circuits and Systems in the Information Age ISCAS '97,* (pp. 1788-1791) .Hong Kong, China: June 12.

[57] Hasan, M. M., Rashid, A. B., & Hussain, M. M. (2010). A novel match-line selective charging scheme for high-speed, low-power and noise-tolerant content- addressable memory. *2010 International Conference on Intelligent and Advanced Systems,* (pp. 1-4). Kuala Lumpur, Malaysia : June 15-17.

[58] Pagiamtzis, K., & Sheikholeslami, A. (2004). A low-power content-addressable memory (CAM) using pipelined hierarchical search scheme. *IEEE Journal of Solid-State Circuits,* 39 (9), 1512-1519.

[59] Karthik, M., Jegan, R. R., & Venkatesan, G. K. D. P. (2019). Content Addressable Memory with Efficient Power Consumption and Throughput. *International Journal of Emerging Trends in Science and Technology IJETST*, 1 (3), 399-404.

[60] Grossi, A., Vianello, E., Zambelli, C., Royer, P., Noel, J., Giraud, B., Perniola, L., Olivo, P., & Nowak, E. (2018). Experimental Investigation of 4-kb RRAM Arrays Programming Conditions Suitable for TCAM. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 26, 2599-2607.

[61] Maxfield, C. M. (2008). *FPGA Architectures*. FPGAs: Instant Access, Massachusetts, USA: Newnes.

[62] Trimberger, S. (2015). Three Ages of FPGAs: A Retrospective on the First Thirty Years of FPGA Technology. *Proceedings of the IEEE*, 103, 318-331.

[63] Abdelhadi, A., Lemieux, G. G., & Shannon, L. (2018). Modular Block-RAM-Based Longest-Prefix Match Ternary Content-Addressable Memories. *28th International Conference on Field Programmable Logic and Applications (FPL),* 243-2437. Dublin, Ireland: August 27-31.

[64] Stimpfling, T., Bélanger, N., Langlois, J. M., & Savaria, Y. (2019). SHIP: A Scalable High-Performance IPv6 Lookup Algorithm That Exploits Prefix Characteristics. *IEEE/ACM Transactions on Networking,* 27(4), 1529-1542.

[65] Xilinx (2021). *Ternary CAM Search LogiCORE IP Product Guide* (PG318). Retrieved from https://docs.xilinx.com/r/2.2-English/pg318-tcam/Introduction

[66] Ullah, Z., Ilgon, K., & Baeg, S. (2012). Hybrid partitioned SRAM-based ternary content addressable memory, *IEEE Transactions on Circuits and Systems I: Regular Papers*, 59 (12), 2969–2979.

[67] Ullah, Z., Jaiswal, M. K., & Cheung, R. C. (2015). Z-TCAM: an SRAM based architecture for TCAM. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 23 (2), 402–406.

[68] Ullah, Z., Jaiswal, M. K., Cheung, R.C. C., & So, H. K. H. (2014). E-TCAM: An Efficient SRAM-Based Architecture for TCAM. *Circuits, Systems, and Signal Processing*, 33 (10), 3123-3144.

[69] Ullah, Z., Jaiswal, M.K., Cheung, R.C., & So, H.K. (2015). UE-TCAM: An ultra efficient SRAM-based TCAM. *TENCON 2015 - IEEE Region 10 Conference*, (pp.1-6). Macao, China: November 1-4.

[70] Ullah, I., Ullah, Z., & Lee, J. (2018). EE-TCAM: An Energy-Efficient SRAM-Based TCAM on FPGA. *Electronics*, 7(9), 186-201.

[71] Jiang, W. (2013). Scalable Ternary Content Addressable Memory implementation using FPGAs. *9th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, (pp.71-82). San Jose, California, USA: October 21-22.

[72] Qian, Z. & Margala, M. (2014). Low power RAM-based hierarchical CAM on FPGA. *International Conference on ReConFigurable Computing and FPGAs (ReConFig14)*, (pp.1-4) . Cancun, Mexico: December 8-10 .

[73] Ullah, I., Ullah, Z., & Lee, J. (2018). Efficient TCAM Design Based on Multi Pumping-Enabled Multi ported SRAM on FPGA. *IEEE Access*, 6, 19940-19947.

[74] Locke K. (2011). *Parameterizable content-addressable memory*, *Xilinx Application Note XAPP1151*. Retrieved from https://docs.xilinx.com/v/u/en-US/xapp1151 _Param _CAM

[75] Xilinx (2017).*Ternary Content Addressable Memory (TCAM) Search IP for SDNet*, Xilinx Product Guide PG190, San Jose, USA. Retrieved from https://www.xilinx.com/support/documents/sw_manuals/xilinx2017_4/UG1012-sdnet-packet- processor.pdf

[76] Irfan, M., Ullah, Z., & Cheung, R. (2019). Zi-CAM: A Power and Resource Efficient Binary Content-Addressable Memory on FPGAs. *Electronics*, 8 (5), 584-596.

[77] Somasundaram, M. (2006). US Patent No. 7,155,563. *Circuits to generate a sequential index for an input number in a pre-defined list of numbers*. DC: U.S. Patent and Trademark Office.

[78] Irfan, M., Ullah, Z., & Cheung, R. C. (2019). D-TCAM: A High-Performance Distributed RAM Based TCAM Architecture on FPGAs. *IEEE Access*, 7, 96060- 96069.

[79] Reviriego, P., Ullah, A., & Pontarelli, S. (2019). PR-TCAM: Efficient TCAM Emulation on Xilinx FPGAs Using Partial Reconfiguration. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 27 (8), 1952-1956.

[80] Ullah, I., Ullah, Z., Afzaal, U., & Lee, J. (2019). DURE: An Energy- and Resource- Efficient TCAM Architecture for FPGAs With Dynamic Updates. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 27 (6), 1298-1307.

[81] Mahmood, H., Ullah, Z., Mujahid, O.M., Ullah, I., & Hafeez, A. (2019). Beyond the Limits of Typical Strategies: Resources Efficient FPGA-Based TCAM. *IEEE Embedded Systems Letters*, 11 (3), 89-92.

[82] Irfan, M. & Ullah, Z. (2017). G-AETCAM: Gate-Based Area-Efficient Ternary Content-Addressable Memory on FPGA. *IEEE Access*, 5, 20785-20790.

[83] Sultana, N., Paira, S., Chandra, S., & Alam, S.S. (2017). A brief study and analysis of different searching algorithms. *Second International Conference on Electrical, Computer and Communication Technologies (ICECCT),* (pp.1-4). Coimbatore, India : February 22-24.

[84] Nowak, R. (2008).Generalized binary search. *46th Annual Allerton Conference on Communication, Control, and Computing,* (pp. 568-574). Monticello, IL, USA: September 23-26 .

[85] Fukac, T. & Korenek, J. (2019). Hash-based Pattern Matching for High Speed Networks. *IEEE 22nd International Symposium on Design and Diagnostics of Electronic Circuits & Systems (DDECS),* (pp. 1-5). Cluj-Napoca, Romania : February 22-24.

[86] Cao, Y., Qi, H., Zhou, W., Kato, J., Li, K., Liu, X., & Gui, J. (2018). Binary Hashing for Approximate Nearest Neighbor Search on Big Data: A Survey. *IEEE Access,* 6, 2039-2054.

[87] Irmayana, A., Sy, H., Paulus, Y. T., Aini, N., & Aryasa, K. B. (2021). A Systematic Comparative Study of Linear, Binary and Interpolation Search Algorithms. *3rd International Conference on Cybernetics and Intelligent System (ICORIS),* (pp.1-5). Makasar, Indonesia: 25-26 October.

[88] Öztekin, H., Temurtaş, F., & Gulbag, A. (2011). A software-based interactive system on BZK.SAU.FPGA10.1 micro computer design for teaching computer architecture and organization. *7th International Conference on Electrical and Electronics Engineering (ELECO),* (pp.II-77-81). Bursa, Turkey: December 1-4.

[89] Terasic Inc. (2008). *DE2 Development and Education Board User Manual*. Retrieved from https://www.terasic.com.tw/attachment/archive/226/DE2_70_ User_manual_v1 05. pdf

[90] Altera, (2011). *Quartus II Design Software.* Retrieved from https://www.farnell.com/datasheets/1536843.pdf

[91] Url-2 <*https://newsroom.intel.com/press-kits/intel-acquisition-of-altera*>, date of access 22/11/2022.

[92] Alioto, M., Consoli, E., & Palumbo, G. (2015). *Flip-Flop Design in Nanometer CMOS: From High Speed to Low Energy*. Cham, Switzerland: Springer.

[93] Roth, J. R., Charles H. (1995). *Fundamentals of Logic Design*. Boston, Massachusetts ,USA :PWS.

[94] Mano, M. M. (1993). *Computer System Architecture*. New Jersey, USA : Prentice Hall.

[95] Altera Corporation (2014). *Internal Memory (RAM and ROM) User Guide*. Retrieved from https://www.intel.com/programmable/technical-pdfs/654378 .pd f

[96] Altera Corporation (2008). *Altera RAM Megafunction User Guide*, Retrieved from https://beamdocs.fnal.gov/AD/DocDB/0077/007799/001/userguide_ram1.pdf

[97] Chaudhary, H. H. (2014). *Data Structures And Algorithms: Made Easy*. New Yor, USA: Programmers Mind LLC.

[98] Schlesinger, R. (2009). *Developing Real World Software.* Sudbury, Massachusetts, USA: Jones & Bartlett Publishers.

[99] Maurer, W. D. & Lewis, T.G. (1975). Hash Table Methods. *ACM Computer survey*, 7, 5-19.

[100] Sipser, M. (2012). *Introduction to the Theory of Computation*. Boston, Massachusetts, USA: Cengage Learning .

[101] Singh, J. P. (2012). *Data Structures And Algorithms Using C*. Noida, India: Vikas Publishing House PVT LTD.

[102] Öztekin, H. (2022). BiCAM-based automated scoring system for digital logic circuit diagrams. *Open Chemistry*, 20(1), 1548-1556.

APPENDICES

Appendix A: Proposed BiCAM VHDL Codes.

- Flip_flop Module :

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity flip_flop is
  Port ( clk: in  std_logic;  Address: in std_logic;
        din : in std_logic; W_R: in std_logic ;
        q: out std_logic   );
end flip_flop;
architecture Behavioral of flip_flop is
begin
process (clk) begin
if rising_edge(clk) then
if  ( W_R= '1' and Address='1')  then
q <= din  ; end if ;
end if ; end process ;
end Behavioral;
```

- BiCAM_BitCell Module :

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity BiCAM_BitCell is
 Port ( clk: in  std_logic;   din : in  std_logic;
       Address : in std_logic  ;  W_R: in std_logic;
       Argument_reg : in std_logic; Key_reg : in std_logic;
       dout: out std_logic ;  Match_Flag : out std_logic );
end BiCAM_BitCell ;
architecture Behavioral of BiCAM_BitCell is
signal reg1: std_logic;
COMPONENT flip_flop
       PORT( clk: in  std_logic; din : in std_logic;
                W_R: in std_logic; Address: in std_logic ;
                q: out std_logic );
END COMPONENT;
signal q: std_logic; begin
BiCAM_BitCell  : flip_flop PORT MAP (clk=>clk,din=>din,W_R=>W_R,Address=>Address,q => q);
process (clk) begin
if rising_edge(clk) then
Match_Flag <= (Argument_reg xnor q )or not( Key_reg) ;
if ( W_R= '0' ) then
dout <= q;end if ;
end if ; end process ;
end Behavioral;
```

- Bitcell_String Module :

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity Bitcell_String is generic ( M : integer:=128; N:integer := 64; adr_b: integer := 8);
Port ( clk: in  std_logic; din : in std_logic_vector (N-1 downto 0 );
       W_R: in std_logic; Address: in std_logic;
       Argument_reg : in std_logic_vector (N-1 downto 0 );
       Key_reg : in std_logic_vector (N-1 downto 0 );
       dout : out std_logic_vector (N-1 downto 0 );
       Match_Flag : out std_logic );
end Bitcell_String;
architecture Behavioral of Bitcell_String is
signal reg1 :std_logic_vector (4 downto 0 ) ;
signal temp: std_logic_vector(4 downto 0);
COMPONENT BiCAM_BitCell
       PORT( clk: in  std_logic; din : in std_logic;
          W_R: in std_logic; Address: in std_logic;
       Argument_reg : in std_logic; Key_reg : in std_logic;
       dout : out std_logic;  Match_Flag: out std_logic);
END COMPONENT;
begin
column : for i in 0 to 4 generate
BiCAM_col  : BiCAM_BitCell PORT mAP
(clk=>clk,Argument_reg=>Argument_reg(i),dout=>dout(i),Address=>Address,W_R=>W_R,Key_reg=>
Key_reg(i),din=>din(i),Match_Flag=>reg1(i));
end generate column  ;
temp(0) <= reg1(0);
gen: for i in 1 to 4 generate
temp(i) <= temp(i-1) and reg1(i);
end generate;
Match_Flag <= temp(4);
end Behavioral;
```

- Mian block Module :

```
use IEEE.STD_LOGIC_1164.ALL;
entity BiCAM is generic ( M : integer := 128 ; N: integer := 64; adr_b: integer := 8);
  Port (  clk: in  std_logic;
          W_R: in  std_logic;
            Mode_Sel:in std_logic;
          reset: in std_logic;
          din : in std_logic_vector (N-1 downto 0 );
          din_Rom : in std_logic_vector (N-1 downto 0 );
          Addres: in std_logic_vector (adr_b-1 downto 0 );
          Adrs_ROM: in std_logic_vector (adr_b-1 downto 0 );
          Key_reg : in std_logic_vector (N-1 downto 0 );
          Argument_reg: in std_logic_vector (N-1 downto 0 );
          BiCAM_out:out std_logic_vector (N-1 downto 0 );
          Machine_code: out std_logic_vector ( 15 downto 0 );
          Match_Flag_adrs : out std_logic_vector (adr_b-1 downto 0 ) );
end BiCAM;
```

```
architecture Behavioral of BiCAM is
signal msel :std_logic_vector ( 127 downto 0 ) ;
type array_reg is Array (0 to M-1) of std_logic_vector (N-1 downto 0) ;
signal CAM_reg:array_reg ;
signal address:std_logic_vector (M-1 downto 0 );
signal adres:std_logic_vector (adr_b-1 downto 0 );
signal adrs_r:std_logic_vector (adr_b-1 downto 0);
signal Match_Flag:std_logic_vector (M-1 downto 0 );
signal data_in:std_logic_vector (N-1 downto 0 );
signal dout:std_logic_vector (N-1 downto 0 );
COMPONENT Bitcell_String
      PORT(  clk: in  std_logic;
           din : in std_logic_vector (N-1 downto 0 );
           dout : out std_logic_vector (N-1 downto 0 );
           W_R: in  std_logic;
           Address: in std_logic;
           Argument_reg : in std_logic_vector (N-1 downto 0 );
           Key_reg : in std_logic_vector (N-1 downto 0 );
           Match_Flag : out std_logic );
END COMPONENT Bitcell_String;
COMPONENT decoder
      PORT(  a : in STD_LOGIC_VECTOR(adr_b-1 downto 0);
           b : out STD_LOGIC_VECTOR(M-1 downto 0) );
END COMPONENT ;
COMPONENT encoder
      PORT(  a : in STD_LOGIC_VECTOR(M-1  downto 0);
           b : out STD_LOGIC_VECTOR(adr_b-1 downto 0) );
END COMPONENT ;
COMPONENT counter
 Port (clk: in  std_logic;
      rst: in  std_logic;
      adrcoun: out std_logic_vector (adr_b-1 downto 0 ));
END COMPONENT ;
begin
Row: for i in 0 to 3 generate
BiCAM_row: Bitcell_String PORT MAP
(clk=>clk,din=>data_in,Argument_reg=>Argument_reg,dout=>CAM_reg(i),W_R=>W_R,Address=>add
ress(i),key_reg=> key_reg,Match_Flag=>Match_Flag(i));
end generate Row  ;
dec: decoder PORT MAP (a=>addres,b=>address);
enc: encoder PORT MAP (a=>Match_Flag,b=>Match_Flag_adrs);
with Mode_Sel select msel<=
Address when '1',
Match_Flag  when '0',
x"00000000000000000000000000000000"  when others;
with reset select data_in <=
 din_Rom when '1',
 din when '0',
x"0000000000000000" when others;
with reset select adres <=
 adrs_ROM  when '1',
 Addres when '0',
 x"00" when others;
with msel select dout <=
```

```
CAM_reg(0)     when x"00000000000000000000000000000001",
CAM_reg(1)     when x"00000000000000000000000000000002",
CAM_reg(2)     when x"00000000000000000000000000000004",
                                .
                                .
CAM_reg(126)   when x"40000000000000000000000000000000",
CAM_reg(127)   when x"80000000000000000000000000000000",
  x"0000000000000000" when others ;
Machine_code <= dout( 15 downto 0 ) ;
BiCAM_out<=dout;
end Behavioral;
```

- Encoder Module :

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity encoder is generic  ( M : integer := 128 ; N: integer := 64; adr_b: integer := 8);
  Port ( a : in STD_LOGIC_VECTOR( 127  downto 0);
        b : out STD_LOGIC_VECTOR(adr_b-1 downto 0));
end encoder;
architecture Behavioral of encoder is
begin
process(a) begin
case a is
when x"00000000000000000000000000000001"=> b <= x"00";
when x"00000000000000000000000000000002"=> b <= x"01";
when x"00000000000000000000000000000004"=> b <= x"02";
                                .
                                .
when x"40000000000000000000000000000000"=> b <= x"7E";
when x"80000000000000000000000000000000"=> b <= x"7F";
when others=>b <=  x"ff";
end case;end process;
end Behavioral;
```

- Decoder Module :

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity decoder is generic  ( M : integer := 128 ; N: integer := 64; adr_b: integer := 8);
  Port (a : in STD_LOGIC_VECTOR(7 downto 0);
        b : out STD_LOGIC_VECTOR(M-1 downto 0) );
end decoder;
architecture Behavioral of decoder is
begin
process(a) begin
case a is
when x"00" => b <= x"00000000000000000000000000000001";
when x"01" => b <= x"00000000000000000000000000000002";
                                .
                                .
when x"7E" => b <= x"40000000000000000000000000000000";
```

```
when x"7F" => b <= x"80000000000000000000000000000000";
when others=> b <=  x"00000000000000000000000000000000";
end case; end process;
end Behavioral;
```

- Delay Module :

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity delay is  generic ( M : integer := 128 ; N: integer := 64; adr_b: integer := 8);
 Port (  clk : in std_logic ;
        din:  in std_logic_vector (adr_b-1 downto 0);
        dout : out std_logic_vector (adr_b-1 downto 0)   ); end delay;
architecture Behavioral of delay is
signal d :std_logic_vector (adr_b-1 downto 0):=x"00";
begin
process(clk)begin
if rising_edge (clk)then
dout<=din
end if ; end process ;
end Behavioral;
```

- Counter Module :

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
entity counter is generic ( M : integer := 128 ; N: integer := 64; adr_b: integer := 8);
  Port (  clk: in  std_logic;
         rst: in  std_logic;
         adrcoun : out std_logic_vector (adr_b-1 downto 0) );
end counter;
architecture Behavioral of counter is
begin
process(clk,rst)
VARIABLE coun : std_logic_vector (adr_b-1 downto 0):= x"00";
begin
if rising_edge (clk)then
if (rst='1')then
if (coun>=x"80") then
coun := x"00";end if ;
coun := coun + "1";
end if ; end if ;
if (coun>0)then
adrcoun <=coun-1;end if ;
adrcoun <=coun; end process;
end Behavioral;
```

- Dec for ROM Module :

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
```

```
entity dec is generic ( M : integer := 128 ; N: integer := 64; adr_b: integer := 8);
  Port (
       din: in std_logic_vector (adr_b-1 downto 0);
       dout : out std_logic_vector (adr_b-2 downto 0) );
end dec;
architecture Behavioral of dec is
begin
dout<=din(adr_b-2 downto 0);
end Behavioral;
```

- BiCAM_128x64 Module :

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
LIBRARY work;
ENTITY BiCAM_128x64 IS
        PORT
        (       clk :  IN  STD_LOGIC; W_R :  IN  STD_LOGIC;
                Mode_Sel :  IN  STD_LOGIC; reset :  IN  STD_LOGIC;
                Addres :  IN  STD_LOGIC_VECTOR(7 DOWNTO 0);
                Argument_reg :  IN  STD_LOGIC_VECTOR(63 DOWNTO 0);
                din :  IN  STD_LOGIC_VECTOR(63 DOWNTO 0);
                Key_reg :  IN  STD_LOGIC_VECTOR(63 DOWNTO 0);
                BiCAM_out :  OUT  STD_LOGIC_VECTOR(63 DOWNTO 0);
                machine_code :  OUT  STD_LOGIC_VECTOR(15 DOWNTO 0);
                Match_Flag_adrs :  OUT  STD_LOGIC_VECTOR(7 DOWNTO 0));
END BiCAM_128x64;
ARCHITECTURE bdf_type OF BiCAM_128x64 IS
COMPONENT mem
        PORT(clock : IN STD_LOGIC;
                address : IN STD_LOGIC_VECTOR(6 DOWNTO 0);
                q : OUT STD_LOGIC_VECTOR(63 DOWNTO 0)  );
END COMPONENT;
COMPONENT counter
GENERIC (adr_b : INTEGER; M : INTEGER; N : INTEGER);
        PORT(clk : IN STD_LOGIC;
                 rst : IN STD_LOGIC;
                 adrcoun : OUT STD_LOGIC_VECTOR(7 DOWNTO 0)
        );
END COMPONENT;
COMPONENT dec
GENERIC (adr_b : INTEGER; M : INTEGER; N : INTEGER);
        PORT(din : IN STD_LOGIC_VECTOR(7 DOWNTO 0);
                 dout : OUT STD_LOGIC_VECTOR(6 DOWNTO 0)
        );END COMPONENT;
COMPONENT delay
GENERIC (adr_b : INTEGER; M : INTEGER; N : INTEGER);
        PORT(clk : IN STD_LOGIC;
                 din : IN STD_LOGIC_VECTOR(7 DOWNTO 0);
                 dout : OUT STD_LOGIC_VECTOR(7 DOWNTO 0)
                 );
END COMPONENT;
COMPONENT BiCAM
GENERIC (adr_b : INTEGER; M : INTEGER; N : INTEGER);
```

```
        PORT(clk : IN STD_LOGIC; W_R : IN STD_LOGIC;
                Mode_Sel : IN STD_LOGIC; reset : IN STD_LOGIC;
                Addres : IN STD_LOGIC_VECTOR(7 DOWNTO 0);
                Adrs_ROM : IN STD_LOGIC_VECTOR(7 DOWNTO 0);
                Argument_reg : IN STD_LOGIC_VECTOR(63 DOWNTO 0);
                din_Rom : IN STD_LOGIC_VECTOR(63 DOWNTO 0);
                din : IN STD_LOGIC_VECTOR(63 DOWNTO 0);
                Key_reg : IN STD_LOGIC_VECTOR(63 DOWNTO 0);
                BiCAM_out : OUT STD_LOGIC_VECTOR(63 DOWNTO 0);
                Machine_code : OUT STD_LOGIC_VECTOR(15 DOWNTO 0);
                Match_Flag_adrs : OUT STD_LOGIC_VECTOR(7 DOWNTO 0)
        );
END COMPONENT;
SIGNAL          SYNTHESIZED_WIRE_0 :  STD_LOGIC_VECTOR(6 DOWNTO 0);
SIGNAL          SYNTHESIZED_WIRE_5 :  STD_LOGIC_VECTOR(7 DOWNTO 0);
SIGNAL          SYNTHESIZED_WIRE_3 :  STD_LOGIC_VECTOR(7 DOWNTO 0);
SIGNAL          SYNTHESIZED_WIRE_4 :  STD_LOGIC_VECTOR(63 DOWNTO 0);
BEGIN
ROM : mem
PORT MAP(clock => clk,address => SYNTHESIZED_WIRE_0,
q => SYNTHESIZED_WIRE_4);
counter0 : counter
GENERIC MAP(adr_b => 8,M => 128,N => 64)
PORT MAP(clk => clk,rst => reset,adrcoun => SYNTHESIZED_WIRE_5);
decoder : dec
GENERIC MAP(adr_b => 8,M => 128,N => 64)
PORT MAP(din => SYNTHESIZED_WIRE_5,
dout => SYNTHESIZED_WIRE_0);
delay0 : delay
GENERIC MAP(adr_b => 8,M => 128,N => 64)
PORT MAP(clk => clk, din => SYNTHESIZED_WIRE_5,
                dout => SYNTHESIZED_WIRE_3);
MainBlock : BiCAM
GENERIC MAP(adr_b => 8,         M => 128, N => 64)
PORT MAP(clk => clk,
                W_R => W_R,   Mode_Sel => Mode_Sel,
                reset => reset,     Addres => Addres,
                Adrs_ROM => SYNTHESIZED_WIRE_3,
                Argument_reg => Argument_reg,
                din_Rom => SYNTHESIZED_WIRE_4,
                din => din, Key_reg => Key_reg,
                BiCAM_out => BiCAM_out,
                Machine_code => machine_code,
                Match_Flag_adrs => Match_Flag_adrs);
END bdf_type;
```

- Prepare Key.reg Module :

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
entity main is
  Port (clk : in std_logic ;
```

```
        dinAssembly: in std_logic_vector (15 downto 0);
        finish_Flag: out std_logic :='0' ;
        adressAssem: out std_logic_vector (5 downto 0):="000000";
        outputAssembly: out std_logic_vector (39 downto 0):= x"0000000000";
        outputCAMkey:out std_logic_vector (63 downto 0):= x"0000000000000000");
end main;
architecture Behavioral of main is
signal outputCAMkey_reg: std_logic_vector (63 downto 0):=x"0000000000000000";
signal outputassembly_reg: std_logic_vector (39 downto 0):=x"0000000000";
signal adressAssem_reg:  std_logic_vector (5 downto 0):="000000";
signal dely : std_logic:='0';
signal count : integer :=0;
signal countAssembly: integer :=0;
signal wai: std_logic:='0';
signal wai2: std_logic:='0';
process (clk)
variable assmbly_flag : std_logic:='0';
variable begining  : std_logic:='0';
variable matchfalgL  : std_logic:='0';
variable matchfalgR  : std_logic:='0';
variable begfalgL  : std_logic:='0';
variable begfalgR  : std_logic:='0';
begin
if rising_edge (clk)then
if (wai='0')then
wai<='1';adressAssem_reg<=adressAssem_reg+"01";end if ;
 if (count<=1) then count <=count+1; end if ;
if(count=2 and wai='1' ) then
outputCAMkey( 63 downto 48 )<= dinAssembly;adressAssem_reg<=adressAssem_reg+"01";
count<=3; finish_Flag<='0';
elsif(count=3) then
outputAssembly(39 downto 0)<=x"0000000000";
if (dinAssembly(15 downto 8)=  x"23" or dinAssembly(15 downto 8)=  x"24" or dinAssembly(15 downto
8)=  x"25" or dinAssembly(15 downto 8)=  x"2A" or dinAssembly(15 downto 8)=  x"40"or
dinAssembly(15 downto 8)=  x"3B") then
if (dinAssembly(15 downto 8)=  x"3B")then
outputCAMkey( 47 downto 32 )<=x"0000";
outputassembly_reg(39 downto 32)<=dinAssembly(7 downto 0);
count <=10 ;countAssembly<=0;assmbly_flag :='1'; matchfalgL:='1';
adressAssem_reg<=adressAssem_reg+"01";
else
outputCAMkey( 47 downto 32 )<= dinAssembly(15 downto 8)& x"00";
outputassembly_reg(39 downto 32)<=dinAssembly(7 downto 0);
count <=10 ;countAssembly<=0;assmbly_flag :='1';
matchfalgL:='1';adressAssem_reg<=adressAssem_reg+"01"; end if ;
elsif (dinAssembly(7 downto 0)=  x"23" or dinAssembly(7 downto 0)=  x"24" or dinAssembly(7 downto
0)=  x"25" or dinAssembly(7 downto 0)=  x"2A" or dinAssembly(7 downto 0)=  x"40" or dinAssembly(7
downto 0)=  x"3B") then
if (dinAssembly(7 downto 0)=  x"3B")then
outputCAMkey( 47 downto 32 )<=dinAssembly(15 downto 8)& x"00";
count <=10 ;countAssembly<=0;assmbly_flag :='1'; matchfalgR:='1';
adressAssem_reg<=adressAssem_reg+"01";
else
count <=10 ; countAssembly<=0;matchfalgR:='1';assmbly_flag :='1';
```

```
outputCAMkey(47 downto 32)<=dinAssembly;
adressAssem_reg<=adressAssem_reg+"01"; end if ;
else
outputCAMkey(47 downto 32)<=dinAssembly;
adressAssem_reg<=adressAssem_reg+"01";
count<=4; end if;
elsif(count= 4 ) then
if (dinAssembly(15 downto 8)=  x"23" or dinAssembly(15 downto 8)=  x"24" or dinAssembly(15 downto
8)=  x"25" or dinAssembly(15 downto 8)=  x"2A" or dinAssembly(15 downto 8)=  x"40"or
dinAssembly(15 downto 8)=  x"3B") then
if (dinAssembly(15 downto 8)=  x"3B")then
outputCAMkey(31 downto 16 )<= x"0000";
outputassembly_reg(39 downto 32)<=dinAssembly(7 downto 0);
count <=10 ;countAssembly<=0;assmbly_flag :='1';
matchfalgL:='1';adressAssem_reg<=adressAssem_reg+"01";
else
outputCAMkey(31 downto 16 )<= dinAssembly(15 downto 8)& x"00";
outputassembly_reg(39 downto 32)<=dinAssembly(7 downto 0);
count <=10 ;countAssembly<=0;assmbly_flag :='1';
matchfalgL:='1';adressAssem_reg<=adressAssem_reg+"01"; end if ;
elsif (dinAssembly(7 downto 0)=  x"23" or dinAssembly(7 downto 0)=  x"24" or dinAssembly(7 downto
0)=  x"25" or dinAssembly(7 downto 0)=  x"2A" or dinAssembly(7 downto 0)=  x"40" or dinAssembly(7
downto 0)=  x"3B") then
if (dinAssembly(7 downto 0)=  x"3B")then
count <=10 ;countAssembly<=0;
outputCAMkey(31 downto 16 )<= dinAssembly(15 downto 8)& x"00";
matchfalgR:='1';assmbly_flag :='1';adressAssem_reg<=adressAssem_reg+"01";
else
count <=10 ;countAssembly<=0;
outputCAMkey(31 downto 16 )<=dinAssembly;
matchfalgR:='1';assmbly_flag :='1';
adressAssem_reg<=adressAssem_reg+"01"; end if ;
else
outputCAMkey(31 downto 16 )<=dinAssembly;
adressAssem_reg<=adressAssem_reg+"01";
count<=5;end if ;
elsif(count= 5) then
if (dinAssembly(15 downto 8)=  x"23" or dinAssembly(15 downto 8)=  x"24" or dinAssembly(15 downto
8)=  x"25" or dinAssembly(15 downto 8)=  x"2A" or dinAssembly(15 downto 8)=  x"40"or
dinAssembly(15 downto 8)=  x"3B") then
if (dinAssembly(15 downto 8)=  x"3B")then
outputCAMkey(31 downto 16 )<= x"0000";
outputassembly_reg(39 downto 32)<=dinAssembly(7 downto 0);
count <=10 ;countAssembly<=0;assmbly_flag :='1';
matchfalgL:='1';adressAssem_reg<=adressAssem_reg+"01";
else
outputCAMkey( 15 downto 0 )<= dinAssembly(15 downto 8)& x"00";
outputassembly_reg(39 downto 32)<=dinAssembly(7 downto 0);
count <=10 ;countAssembly<=0;assmbly_flag :='1';
matchfalgL:='1';adressAssem_reg<=adressAssem_reg+"01"; end if ;
elsif (dinAssembly(7 downto 0)=  x"23" or dinAssembly(7 downto 0)=  x"24" or dinAssembly(7 downto
0)=  x"25" or dinAssembly(7 downto 0)=  x"2A" or dinAssembly(7 downto 0)=  x"40" or dinAssembly(7
downto 0)=  x"3B") then
if (dinAssembly(7 downto 0)=  x"3B")then
```

```
count <=10 ;countAssembly<=0;outputCAMkey( 15 downto 0 )<= dinAssembly(15 downto 8)& x"00";
matchfalgR:='1';assmbly_flag :='1';
adressAssem_reg<=adressAssem_reg+"01";
else
count <=10 ;countAssembly<=0;outputCAMkey (15 downto 0)<=dinAssembly;
matchfalgR :='1';assmbly_flag :='1';adressAssem_reg<=adressAssem_reg+"01"; end if ;
else
outputCAMkey (15 downto 0)<=dinAssembly;
adressAssem_reg<=adressAssem_reg+"01";count<= 2;
assmbly_flag:='0';end if ;
elsif (count=10) then
finish_Flag<='1';
if ( matchfalgR ='1' ) then
if (countAssembly = 0 ) then
outputAssembly(39 downto 24)<=dinAssembly;
adressAssem_reg<=adressAssem_reg+"01";countAssembly<=1 ;
elsif (countAssembly = 1) then
outputCAMkey<=x"0000000000000000";outputAssembly(23 downto 8)<=dinAssembly;
adressAssem_reg<=adressAssem_reg+"01";countAssembly<=2;
elsif (countAssembly =2 ) then
outputAssembly(7 downto 0 )<=dinAssembly(15 downto 8);
adressAssem_reg<=adressAssem_reg+"01";
count <= 2; matchfalgR :='0'; end if ;
 elsif (matchfalgL ='1' ) then
if (countAssembly = 0 ) then
outputAssembly(31 downto 16)<=dinAssembly;
adressAssem_reg<=adressAssem_reg+"01";
countAssembly<=1;
elsif (countAssembly = 1) then
outputCAMkey<=x"0000000000000000";
outputAssembly(15 downto 0 )<=dinAssembly;
adressAssem_reg<=adressAssem_reg+"01";
countAssembly<=2;
elsif (countAssembly = 2) then
outputCAMkey(63 downto 56 )<=dinAssembly(7 downto 0);
count <= 21;
adressAssem_reg<=adressAssem_reg+"01";
matchfalgL :='0';
 end if ;end if ;
  elsif (count=21) then
 finish_Flag<='0';
 outputCAMkey( 55 downto 40 )<= dinAssembly;
adressAssem_reg<=adressAssem_reg+"01";
count<=22;
 elsif (count=22) then
outputAssembly(39 downto 0)<=x"0000000000";
 if (dinAssembly(15 downto 8)=  x"23" or dinAssembly(15 downto 8)=  x"24" or dinAssembly(15
downto 8)=  x"25" or dinAssembly(15 downto 8)=  x"2A" or dinAssembly(15 downto 8)=  x"40"or
dinAssembly(15 downto 8)=  x"3B") then
if (dinAssembly(15 downto 8)=  x"3B")then
outputCAMkey( 39 downto 24 )<= x"0000";
outputassembly_reg(39 downto 32 )<=dinAssembly(7 downto 0);
count <=10 ;countAssembly<=0;assmbly_flag :='1';
matchfalgL:='1';adressAssem_reg<=adressAssem_reg+"01"; else
```

```
outputCAMkey( 39 downto 24 )<= dinAssembly(15 downto 8)& x"00";
outputassembly_reg(39 downto 32 )<=dinAssembly(7 downto 0);
count <=10 ;countAssembly<=0;assmbly_flag :='1';
matchfalgL:='1';adressAssem_reg<=adressAssem_reg+"01";  end if ;
elsif (dinAssembly(7 downto 0)=  x"23" or dinAssembly(7 downto 0)=  x"24" or dinAssembly(7 downto
0)=  x"25" or dinAssembly(7 downto 0)=  x"2A" or dinAssembly(7 downto 0)=  x"40" or dinAssembly(7
downto 0)=  x"3B") then
if (dinAssembly(7 downto 0)=  x"3B")then
count <=10 ; countAssembly<=0;outputCAMkey( 39 downto 24)<=dinAssembly(15 downto 8)& x"00";
matchfalgR:='1';assmbly_flag :='1';adressAssem_reg<=adressAssem_reg+"01";
else
count <=10 ; countAssembly<=0;outputCAMkey( 39 downto 24)<=dinAssembly;
matchfalgR:='1';assmbly_flag :='1';adressAssem_reg<=adressAssem_reg+"01"; end if ;
else
outputCAMkey( 39 downto 24)<=dinAssembly;adressAssem_reg<=adressAssem_reg+"01";
count<=23 ; end if;
elsif (count=23) then
 if (dinAssembly(15 downto 8)=  x"23" or dinAssembly(15 downto 8)=  x"24" or dinAssembly(15
downto 8)=  x"25" or dinAssembly(15 downto 8)=  x"2A" or dinAssembly(15 downto 8)=  x"40"or
dinAssembly(15 downto 8)=  x"3B") then
if (dinAssembly(15 downto 8)=  x"3B")then
outputCAMkey( 39 downto 24 )<= x"0000";
outputassembly_reg( 39 downto 32 )<=dinAssembly(7 downto 0);
count <=10 ;countAssembly<=0;assmbly_flag :='1';
matchfalgL:='1';adressAssem_reg<=adressAssem_reg+"01";
else
outputCAMkey( 7 downto 0 )<= dinAssembly(15 downto 8);
outputassembly_reg( 39 downto 32 )<=dinAssembly(7 downto 0);
count <=10 ;countAssembly<=0;assmbly_flag :='1'; matchfalgL:='1';
adressAssem_reg<=adressAssem_reg+"01"; end if ;
elsif (dinAssembly(7 downto 0)=  x"23" or dinAssembly(7 downto 0)=  x"24" or dinAssembly(7 downto
0)=  x"25" or dinAssembly(7 downto 0)=  x"2A" or dinAssembly(7 downto 0)=  x"40" or dinAssembly(7
downto 0)=  x"3B") then
if (dinAssembly(7 downto 0)=  x"3B")then
outputCAMkey( 39 downto 24)<=dinAssembly(15 downto 8)& x"00";
count <=10 ;countAssembly<=0; matchfalgR:='1';
assmbly_flag :='1';adressAssem_reg<=adressAssem_reg+"01";
else
count <=10 ;countAssembly<=0; matchfalgR:='1';
assmbly_flag :='1';adressAssem_reg<=adressAssem_reg+"01"; end if ;
else
outputCAMkey( 7 downto 0 )<= dinAssembly(15 downto 8);
adressAssem_reg<=adressAssem_reg+"01";
count<=21 ; end if;  end if ;
adressAssem<=adressAssem_reg;
end if ;end process ;
end Behavioral;
```

- Top _ key_reg Module:

```
library IEEE; LIBRARY ieee;
USE ieee.std_logic_1164.all;
LIBRARY work;
```

```
ENTITY top IS   PORT    (clk :  IN  STD_LOGIC; finish_Flag :  OUT  STD_LOGIC;
                  output :  OUT  STD_LOGIC_VECTOR(63 DOWNTO 0);
                  outputAssembly :  OUT  STD_LOGIC_VECTOR(39 DOWNTO 0));
END top;
ARCHITECTURE bdf_type OF top IS
COMPONENT main
        PORT(clk : IN STD_LOGIC;
                 dinAssembly : IN STD_LOGIC_VECTOR(15 DOWNTO 0);
                 finish_Flag : OUT STD_LOGIC;              adressAssem : OUT
STD_LOGIC_VECTOR(5 DOWNTO 0);
                 outputAssembly : OUT STD_LOGIC_VECTOR(39 DOWNTO 0);
                 outputCAMkey : OUT STD_LOGIC_VECTOR(63 DOWNTO 0));
END COMPONENT;
COMPONENT ro
        PORT(clock : IN STD_LOGIC;
                 address : IN STD_LOGIC_VECTOR(5 DOWNTO 0);
                 q : OUT STD_LOGIC_VECTOR(15 DOWNTO 0)  );
END COMPONENT;
SIGNAL          adressAssem :  STD_LOGIC_VECTOR(5 DOWNTO 0);
SIGNAL          SYNTHESIZED_WIRE_0 :  STD_LOGIC_VECTOR(15 DOWNTO 0);
BEGIN
b2v_inst : main
PORT MAP(clk => clk,
                 dinAssembly => SYNTHESIZED_WIRE_0,
                 finish_Flag => finish_Flag,
                 adressAssem => adressAssem,
                 outputAssembly => outputAssembly,
                 outputCAMkey => output);
b2v_inst1 : ro
PORT MAP(clock => clk,address => adressAssem,  q => SYNTHESIZED_WIRE_0);
END bdf_type;
```

Appendix B: VHDL Codes of RAM-based Brute-force Algorithm

- Main Module:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
entity main is
 port (  clk : in std_logic ;
       dinAssembly: in std_logic_vector ( 15 downto 0);
       dindata: in std_logic_vector (15 downto 0);
       adresAssembly:out std_logic_vector (5 downto 0):="000000";
       adresdata:out std_logic_vector (15 downto 0):=x"0000";
       outputAssembly: out  std_logic_vector (39 downto 0):=x"0000000000";
       outputmachinecode:out std_logic_vector (15 downto 0):=x"0000" );
end main ;
architecture Behavioral of main is
signal wai: std_logic:='0';
```

```
signal numoutput:    std_logic_vector (15 downto 0):=x"0000";
signal matchUpper: std_logic:='0';signal matchLower: std_logic:='0';
signal match: std_logic:='0'; signal coun : std_logic_vector (1 downto 0):="00";
begin
process (clk)
variable regAdress: std_logic_vector (5 downto 0):="000000";
variable flagUpper: std_logic:='0';variable flagnumR: std_logic:='0';
variable flagnumL: std_logic:='0';variable flagLower: std_logic:='0';
variable reg: std_logic:='0';variable varmatch: std_logic:='0';
variable RadresAssembly: std_logic_vector (5 downto 0):="000000";
variable Radresdata:    std_logic_vector (15 downto 0):=x"0000";
begin
if rising_edge (clk) then
wai<='1';
if (wai='1')then
if (dinAssembly (15 downto 8) = dindata(15 downto 8)) then
if(dinAssembly(15 downto 8)=  x"23" or dinAssembly(15 downto 8 )=  x"24" or dinAssembly(15 downto
8 )=  x"25" or dinAssembly( 15 downto 8)=  x"2A" or dinAssembly(15 downto 8)=  x"40" or
dinAssembly(15 downto 8)=  x"3B") then
outputmachinecode(15 downto 8)<= dindata (7 downto 0);
flagnumL:='1';match <='1';matchUpper<='1';varmatch:='1';end if ;end if ;
if (match ='1') then
if (matchUpper ='1')then
outputmachinecode(7 downto 0)<= dindata (15 downto 8);
elsif (matchlower ='1')then
outputmachinecode(15 downto 0)<= dindata (15 downto 0);end if ;
match<='0';reg:='1';end if ;
if (reg = '1') then
if (flagnumR ='1') then
if (coun="00") then
outputAssembly(39 downto 24 )<= dinAssembly(15 downto 0);
RadresAssembly:=RadresAssembly+"01";coun<=coun+"01";
elsif (coun="01") then
outputAssembly(23 downto 8)<= dinAssembly(15 downto 0);
RadresAssembly:=RadresAssembly+"01";coun<=coun+"01";
elsif (coun="10") then
outputAssembly(7 downto 0 )<= dinAssembly(15 downto 8);flagnumR:='0';end if ;
elsif (flagnumL ='1' ) then
if (coun="00") then
outputAssembly(39 downto 32 )<= dinAssembly(7 downto 0);
RadresAssembly:=RadresAssembly+"01";coun<=coun+"01";
elsif (coun="01") then
outputAssembly(31 downto 16)<= dinAssembly(15 downto 0);
RadresAssembly:=RadresAssembly+"01";
coun<=coun+"01";
elsif (coun="10") then
outputAssembly(15 downto 0 )<= dinAssembly(15 downto 0);
regAdress:=RadresAssembly+"01"; coun<=coun+"01";
elsif (coun="11") then
RadresAssembly:=RadresAssembly+"01";
flagnumL:='0';end if ;
if (dinAssembly (15 downto 8) = x"3B") then
regAdress:=RadresAssembly;
Radresdata:= x"0000";reg := '0';matchUpper <='0';matchlower <='0';
```

```
elsif (dinAssembly (7 downto 0) = x"3B")then
RadresAssembly:=RadresAssembly+"01";
regAdress:=RadresAssembly;Radresdata:= x"0000";reg := '0';
matchUpper <='0';matchlower <='0';end if ;end if ;
elsif (reg='0')then
coun<="00";
if ((dinAssembly (15 downto 8)= x"20" or dinAssembly (15 downto 8)= x"3B")  and match ='0') then
if (dinAssembly (15 downto 8) = x"3B" ) then
outputmachinecode<= dindata ;flagnumL:='1';reg:='1';
elsif (dinAssembly (7 downto 0)= dindata(15 downto 8) ) then
if(dinAssembly(7 downto 0)=  x"23" or dinAssembly(7 downto 0)=  x"24" or dinAssembly(7 downto 0)=
x"25" or dinAssembly(7 downto 0)=  x"2A" or dinAssembly(7 downto 0)=  x"40" ) then
outputmachinecode(15 downto 8)<= dindata (7 downto 0);
RadresAssembly:=RadresAssembly+"01"; Radresdata:= Radresdata + "01";
flagnumR:='1';matchUpper<='1';match <='1';
end if ;
elsif (dinAssembly (7 downto 0)= dindata(7 downto 0) )then
if(dinAssembly(7 downto 0)=  x"23" or dinAssembly(7 downto 0)=  x"24" or dinAssembly(7 downto 0)=
x"25" or dinAssembly(7 downto 0)=  x"2A" or dinAssembly(7 downto 0)=  x"40" ) then
Radresdata:= Radresdata + "01";
RadresAssembly:=RadresAssembly+"01"; flagnumR:='1'; matchLower<='1'; match <='1';
else
Radresdata:= Radresdata+"01"; end if ;
else
RadresAssembly := regAdress ;Radresdata:= Radresdata+"01";end if ;
elsif (dinAssembly (7 downto 0)= x"20"  or dinAssembly (7 downto 0)= x"3B" ) then
if(dindata(7 downto 0)=  x"23" or dindata(7 downto 0)=  x"24" or dindata(7 downto 0)=  x"25" or
dindata(7 downto 0)=  x"2A" or dindata(7 downto 0)=  x"40" ) then
RadresAssembly:=RadresAssembly+"01";
else
RadresAssembly:=RadresAssembly+"01";
Radresdata:= Radresdata + "01";end if ;
elsif (dinAssembly (15 downto 8) = dindata(7 downto 0) )then
if(dinAssembly(15 downto 8)=  x"23" or dinAssembly(15 downto 8 )=  x"24" or dinAssembly(15 downto
8 )=  x"25" or dinAssembly( 15 downto 8)=  x"2A" or dinAssembly(15 downto 8)=  x"40" ) then
matchLower<='1';match <='1';flagnumL:='1';end if ;
Radresdata:= Radresdata + "01";
elsif (dinAssembly (7 downto 0) = dindata(15 downto 8)) then
if(dinAssembly(7 downto 0)=  x"23" or dinAssembly(7 downto 0)=  x"24" or dinAssembly(7 downto 0)=
x"25" or dinAssembly(7 downto 0)=  x"2A" or dinAssembly(7 downto 0)=  x"40" ) then
outputmachinecode(15 downto 8)<= dindata (7 downto 0);
Radresdata:= Radresdata + "01";flagnumL:='1';matchUpper<='1';match <='1';end if ;
RadresAssembly:=RadresAssembly+"01"; flagUpper:='1';
elsif (dinAssembly ( 7 downto 0) = dindata(7 downto 0) ) then
if(dinAssembly(7 downto 0)=  x"23" or dinAssembly(7 downto 0)=  x"24" or dinAssembly(7 downto 0)=
x"25" or dinAssembly(7 downto 0)=  x"2A" or dinAssembly(7 downto 0)=  x"40" ) then
matchLower<='1';match <='1'; end if ;
RadresAssembly:=RadresAssembly+"01"; Radresdata:= Radresdata + "01";
elsif (varmatch ='1') then
Radresdata:= Radresdata+ "01";varmatch :='0';
else
RadresAssembly := regAdress ; Radresdata:= Radresdata+ "01"; end if ;end if ;
wai<='0';
adresAssembly<=RadresAssembly;
```

```
adresdata<= Radresdata;
end if ;end if ;
end process ;
end Behavioral;
```

- Top Module:

```
library IEEE; LIBRARY ieee;
USE ieee.std_logic_1164.all;
LIBRARY work;
ENTITY top IS   PORT    (clk :  IN  STD_LOGIC;
                outputAssembly :  OUT  STD_LOGIC_VECTOR(39 DOWNTO 0);
                outputmachcode :  OUT  STD_LOGIC_VECTOR(15 DOWNTO 0));
END top;
ARCHITECTURE bdf_type OF top IS
COMPONENT romdata
        PORT(clock : IN STD_LOGIC;
                address : IN STD_LOGIC_VECTOR(10 DOWNTO 0);
                q : OUT STD_LOGIC_VECTOR(15 DOWNTO 0)  );
END COMPONENT;
COMPONENT main
        PORT(clk : IN STD_LOGIC;
                dinAssembly : IN STD_LOGIC_VECTOR(15 DOWNTO 0);
                dindata : IN STD_LOGIC_VECTOR(15 DOWNTO 0);
                adresAssembly : OUT STD_LOGIC_VECTOR(5 DOWNTO 0);
                adresdata : OUT STD_LOGIC_VECTOR(15 DOWNTO 0);
                outputAssembly : OUT STD_LOGIC_VECTOR(39 DOWNTO 0);
                outputmachinecode : OUT STD_LOGIC_VECTOR(15 DOWNTO 0) );
END COMPONENT;
COMPONENT romasse
        PORT(   clock : IN STD_LOGIC;
                address : IN STD_LOGIC_VECTOR(5 DOWNTO 0);
                q : OUT STD_LOGIC_VECTOR(15 DOWNTO 0)  );
END COMPONENT;
SIGNAL          adresAssemby :  STD_LOGIC_VECTOR(5 DOWNTO 0);
SIGNAL          adresdata :  STD_LOGIC_VECTOR(15 DOWNTO 0);
SIGNAL          SYNTHESIZED_WIRE_0 :  STD_LOGIC_VECTOR(15 DOWNTO 0);
SIGNAL          SYNTHESIZED_WIRE_1 :  STD_LOGIC_VECTOR(15 DOWNTO 0);
BEGIN
b2v_inst1 : romdata
PORT MAP(clock => clk,address => adresdata(10 DOWNTO 0),q => SYNTHESIZED_WIRE_1);
b2v_inst2 : main
PORT MAP(clk => clk,
                dinAssembly => SYNTHESIZED_WIRE_0,
                dindata => SYNTHESIZED_WIRE_1,
                adresAssembly => adresAssemby,
                adresdata => adresdata,
                outputAssembly => outputAssembly,
                outputmachinecode => outputmachcode);
b2v_RomAssembly : romasse
PORT MAP(clock => clk,address => adresAssemby, q => SYNTHESIZED_WIRE_0);
END bdf_type;
```