

Multi-valued Logic System in Computer Architecture

by

Md. Mostofa Akbar

Submitted to the Department of Computer Science and Engineering
in partial fulfillment of the degree
of
Master of Science in Engineering(CSE)



**Department of Computer Science and Engineering
Bangladesh University of Engineering and Technology**

October, 1998.



Multi-valued Logic System in Computer Architecture

A Thesis

Submitted by

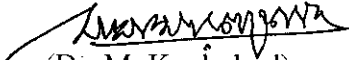
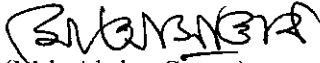


Md. Mostofa Akbar

Roll No.: 9405002P, Session: 1993-94-95

For the partial fulfillment of the degree of
M.Sc. Engg. in Computer Science and Engineering

Examination held on October 04, 1998

Approved as to style and contents by:

1. 
(Dr. M. Kaykobad) (Supervisor) Chairman
Professor and Head,
CSE Department, BUET
2. 
(Md. Abdus Sattar) Member
Assistant Professor
CSE Department, BUET
3. 
(Dr. Md. Abul Kashem Mia) Member
Assistant Professor
CSE Department, BUET
4. 
(Dr. M. Rezwan Khan) Member
Professor (External)
EEE Department, BUET

To

My Friend

M. Mamunul Islam

A real computer scientist I have ever seen.

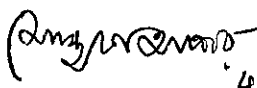
Acknowledgement

Profound knowledge and keen interest of Dr. M. Kaykobad in the fields of Algorithm and specially his enthusiasm of using 3 instead of 2 as algorithmic parameter has influenced the author to carry out the research work in these fields. The research work was done under his kind supervision. His endless patience, scholarly guidance, constant and energetic supervision, valuable advice, suggestions and encouragement at all stages have made it possible to complete the thesis.

The author feels pleased to have the opportunity of expressing his most sincere gratitude to all who helped him in course of his research work.

Special gratitude to author's friend Mr. M. Mamunul Islam, student of M.Sc., University of Texas at Arlington and ex Lecturer of Department of CSE, BUET for his kind cooperation of research work.

The all out support, cooperation and services rendered by the faculty members and staff of the Department of CSE, BUET are also acknowledged with sincere thanks.


(Md. Mostofa Akbar) 4/10/98

Abstract

Computers work on binary principle. The number system, basic operators, logic circuits are all binary. In the thesis a new basic principle of computing systems has been investigated. The architecture of the computer's computing part is such that it can do operations taking two operands at a time. The algorithms that use binary divide and conquer approach use the facility of this binary architecture fully. Sufficient improvement has been attained applying binary divide and conquer approach in traditional algorithms. Binary search and quick-sort are the examples of such algorithms. k -nary search algorithm and quick-sort with k -partitions use divide and conquer approach dividing the problem into k partitions. The extensively used part of computer by these algorithms is comparator. It seems to be more efficient for these algorithms to use a comparator which can compare one element with more than two elements. Three comparator circuits have been proposed here that work in multi-valued fashion. The performance analysis of binary search and quick-sort algorithms has been presented for that comparator architecture. The analysis shows that the performance of quick-sort algorithm has improved from $O(n \log_2 n)$ to $O(n \log_k n)$ with the introduction of new architectures. To achieve this improved performance on required number of comparisons we have to make some additional data transfer operations. The estimated time requirements for compare instruction by proposed comparators have been calculated in terms of unit time. The total cost of k -nary search algorithm and quick-sort algorithm with k partitions has been calculated in terms of time requirement. This cost has been minimized and optimal values of k for different architectures have also been determined. It has been found that whatever hardware have been used k -nary search algorithm shows inferior performance compared to binary search algorithm. But substantial improvement with respect to time requirements have been found for quick-sort algorithm with k -partitions using different proposed hardware for multi-valued comparator.

Table of Contents

Chapter 1	1
Introduction.....	1
1.1 Literature Review.....	1
1.2 Significance of the Topic	2
1.3 Scope and objective of the Thesis.....	2
1.4 Methodology used in the Research.....	3
1.5 Organization of the Thesis	4
Chapter 2.....	5
Preliminaries	5
2.1 Binary Number System.....	5
2.2 Other Popular Number Systems.....	5
2.3 k-nary Number System	6
2.4 Memory Storage for k-nary Multi-valued Number System.....	6
2.5 Binary Logic Gates	7
2.6 Concept of Multi-valued Logic Gates.....	9
2.7 Important Binary Digital Logic Circuits.....	9
2.8 Binary Architecture of ALU	11
2.9 Multi-valued Architecture of ALU	12
Chapter 3.....	13
Algorithms using binary Concept.....	13
3.1 Divide and Conquer Approach of Problem Solving.....	13
3.2 Algorithms using Divide and Conquer Approach	14
3.3 Binary Search Algorithms.....	15
3.4 Quick-sort Algorithm.....	16
3.5 Performance Analysis of Quick-sort Algorithm.....	17
3.6 Need for Multi-valued Architectures	18
Chapter 4.....	19
Proposed Multi-valued Comparator Circuits.....	19
4.1 Binary comparison	19
4.2 Multi-valued Comparison	19

4.3	Current Comparator Circuits used in Computers.....	20
4.4	Proposed Multi-valued Comparator Circuits.....	22
4.5	Comparator by Bit comparison.....	22
4.5.1	H/W to implement this type of comparator.....	22
4.5.2	Comparator of bits between v_i and $A_i^{(i)}$	22
4.5.3	Position counter.....	24
4.5.4	Bit Counter.....	25
4.5.5	Output Available Signal.....	25
4.5.6	Registers Holding the array.....	26
4.5.7	Value Register.....	27
4.6	Parallelism in Comparison.....	27
4.7	Comparator Using Pipelining Idea.....	29
4.8	Time Requirement in three proposed Architecture for comparison.....	31
4.8.1	Time Requirement by Traditional Binary Comparison.....	32
4.8.2	Time requirement of Comparator by doing Bit comparison.....	32
4.8.3	Time requirement of Parallel Comparator.....	33
4.8.4	Time requirement for Comparators with pipelining.....	34
Chapter 5	35
Multi-valued k -nary Search	35
5.1	Algorithm of multi-valued k -nary Search.....	35
5.2	Cost Requirement using Binary Comparator.....	36
5.3	Cost Requirement using Comparators by Bit comparison.....	37
5.4	Cost Requirement using Parallel comparator.....	38
5.5	Cost Requirement using pipelined comparator.....	38
5.6	Comparison among the optimal performance of different comparators for k -nary searching algorithm.....	39
Chapter 6	40
Quick-sort with k partitions	40
6.1	Algorithm of Quick-sort with k partitions.....	40
6.2	Partitioning algorithm.....	42
6.3	Complexity of Multi-valued Quick-sort Algorithm.....	44
6.4	Time Requirement in Quick-sort with k -partitions.....	47

6.4.1	Time requirement of Quick-sort with k-partitions using Binary comparator	47
6.4.2	Time requirement of Quick-sort with k-partitions using bit comparison	48
6.4.3	Time requirement of Quick-sort with k-partitions using parallel comparator	49
6.4.4	Time requirement of Quick-sort with k-partitions using pipelining comparator	49
6.5	Optimal degree for Different Comparator Architectures in Quick-sort Algorithm	50
Chapter 7		53
Experimental Results		53
7.1	Environment of the Experiment	53
7.2	How was the experiment done?	53
7.3	Experimental Results	55
Chapter 8		58
Further Improvement		58
8.1	Shortcomings of the thesis	58
8.2	Future Research Plan	59
8.2.1	New Device to implement ternary logic	59
8.2.2	True Multi-valued Architecture	59
8.2.3	VLSI design of ternary memory cell	59
8.2.4	Double Precision Floating Point Comparator	60
8.2.5	Detailed circuit diagram	60
8.2.6	Cost Benefit Analysis	60
8.2.7	Exact Simulation of Multi-valued Comparator	60
8.2.8	Perfect Analysis of Multi-valued Quick-sort using multi-valued comparator	61
References		62

List of Tables

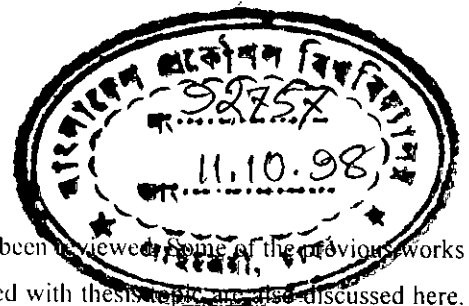
Table 1 Truth table for NAND gate.....	8
Table 2 Truth table for AND gate.....	8
Table 3 Truth table for OR gate.....	8
Table 4 Truth table for NOT gate.....	8
Table 5 Truth table for ternary NOT gate.....	9
Table 6 Truth table of basic block of Comparator.....	21
Table 7 Truth table of final output of Comparator.....	21
Table 8 Truth table for finding OF, RIO,BIO.....	23
Table 9 Time requirement in different steps of pipelined comparator.....	34
Table 10 Complexity of k -nary search for different multi-valued comparator.....	39
Table 11 Optimal values of k for different multi-valued comparators.....	50
Table 12 Ratio of time requirement for multi-valued architecture with respect to binary quick-sort with binary architecture.....	51
Table 13 Time requirement of quick-sort algorithm for different multi-valued architecture.....	55
Table 14 Time requirement of k -nary search algorithm for different multi-valued architecture.....	57

List of Figures

Figure 1 Block diagram of 2 input basic logic gates.	8
Figure 2 Block diagram of Traditional Adder/Subtractor.....	9
Figure 3 Block diagram of registers.....	10
Figure 4 Block diagram of decoder.	11
Figure 5 Block Diagram of binary ALU.....	11
Figure 6 Conceptual block diagram of multi-valued ALU.....	12
Figure 7 Detailed Block diagram of comparator.	20
Figure 8 Circuit diagram of basic block of currently used comparators.....	21
Figure 9 Block Diagram of the proposed comparator unit by bit comparison.	23
Figure 10 Block diagram of Output Ready signal generator.	25
Figure 11 Timing diagram of Clock and Output Ready signal.....	26
Figure 12 Block diagram of Register Array	26
Figure 13 Single unit comparator	27
Figure 14 Block diagram of parallel comparator.....	28
Figure 15 Block diagram of position detector	28
Figure 16 Pipe line Procesor.....	30

Chapter 1

Introduction



This chapter is the review of the thesis. In the first section some literatures have been reviewed. Some of the previous works have been mentioned in this section. The achievements of research works related with thesis topic are also discussed here. In the subsequent sections the objective and scope of the research have been described in a nutshell. The methods of the research and the techniques to be applied in our research have also been mentioned in the section just before last section.

1.1 Literature Review

The primitive computers were all analog computers. These computers were able to give multi level outputs. Not only computers in old age but also the electronic devices of current age are divided into two types - analog devices and digital devices. Analog computers are obsolete now. The main reason of this is the complexity of analog circuitry. It is very difficult to design analog circuits and the complexity of stable analog devices is very high. The invention of transistors and bistable electronic circuits has facilitated scientists to design binary computers and electronic circuits. There is no doubt that Binary circuits are very easy and very stable for designing very sophisticated and challenging machines like computers. The architectures and concepts of operators in the instructions are also based on the binary concepts. The reputation of binary systems has inspired the scientists to explore algorithms using binary logic rather than multi-valued logic. In different algorithms for problem solving binary concept as a problem solving strategy has been used successfully. For example Binary Search, Quick-sort and Heap-sort are some of the examples. Using binary concept in Divide and Conquer Approach problems can be solved more efficiently than by any trivial approach. But some of the papers published in journals show that if we proceed one step ahead and use the less traditional concept i.e., k -nary concept ($k = 3, 4, \dots$) then very good results can be achieved. For example SCHAFFER [12] shows the improvement of Heap-sort algorithm using ternary trees. Similarly GOBEL [3] shows the advantages of ternary trees and KAYKOBAD [7] shows the superiority of 3 over 2 as algorithmic parameter. Actually this is a current trend to use 3 rather than 2 as parameter in problem solving. In computer hardware the ternary systems for memory and error detection code have been proved more efficient but it is yet not possible to implement because such stable hardware is not available. PINTER [11] shows how ternary trees are used in VLSI arrays and MEGIDDO [10] shows inferiority of binary encoding for problem language relationship. Enthusiasm of multi-valued architecture could not flourish due to non availability of suitable electronic devices. So very few papers have been found related to this topic. Lack of suitable multi-valued architecture and device is a hurdle

to apply k -nary algorithms with full efficiency. So it may be worthwhile to design multi-valued architecture using binary logic systems which can improve the performance substantially. That is, a pseudo multi-valued system can serve the purpose to some extent. It may seem going back to the old age of analog computers but this research demonstrates the very stable multi-valued system which may be proved efficient at the same time.

1.2 Significance of the Topic

The topic of the thesis is “Multi-valued Logic System in Computer Architecture”. As mentioned in the previous section the logic system of the computer is 2-valued or binary. There are only two states in electronic circuits. One state is 0V and the other is 5V. The modern computers are based on these two states of electrical voltage. We can say that the architecture of computer uses this binary concept. Current instruction sets i.e., addition, multiplication, subtraction, comparison are all binary. So the architectures to support these operations are also binary. The title signifies the idea of replacing binary system with multi-valued ones. This is a concept of a new architecture which can support multi-valued addition, subtraction and comparison. Thus a wide range of research scope in computer architecture will be discussed in the topic.

1.3 Scope and objective of the Thesis

The scope of the thesis is to study multi-valued architectures in computer system. As the computer is a problem solving machine it also includes study of multi-valued algorithmic parameter for problem solving method. Proposing a new architecture of computer systems or one of the portion of the computing unit of computer as well as reviewing existing research are the main theme of the thesis. The objective of the thesis together with its scope is described below with mentioning its scope.

Study of so far invented architectures and algorithms: At first some of the architectures and algorithms using binary logic will be discussed. There are different algorithms using multi-valued logic system. The performance of these architectures and algorithms will be observed. The performance of the binary algorithms and architectures must be discussed for comparison and justification of the validity of multi-valued architecture.

Proposed Architecture: Three architectures will be mentioned with detailed block diagrams for multi-valued architecture. These architectures may use the binary logic circuit but the concept of the architectures are multi-valued. The relative time requirement of multi-valued operation will be evaluated analytically with some assumption to compare it with binary one.

Proposed Change in Algorithms: Some algorithms will be proposed by modifying the algorithms which use binary concept to use the extra benefit of multi-valued architecture. The detailed analysis will be made to determine the cost of the algorithms using new architectures. The cost of the different new algorithms will be compared with the cost of currently used algorithms.

Experimental Results: As our computers use binary architectures it is not possible to determine the actual time requirement of the algorithms. So it can be determined by simulating the architecture in programs or using some empirical data and equations found from experiments or on the basis of proposed architectures.

1.4 Methodology used in the Research

The main research carried out in the thesis is to design a multi-valued architecture for comparator circuit. To design the architecture the idea of simplicity has been employed. To design such an architecture is not less complex than designing any processors. So special care has been taken to design the proposed architecture. Timing of the circuit has been carefully maintained. Actually these are sequential circuits. Pipelining idea for processor design and parallel processing concept have also been applied in designing the proposed architecture.

To determine the time requirement of a particular architecture the processing flow of any processing unit has been examined carefully. Then the average time requirement of the architecture has been calculated by summing up all the time components of the processing flow. This is determined in terms of a particular unit time, which may be a clock tick time or data transfer time from one location to another location of the memory.

Designing algorithms using multi-valued architecture is not very difficult. For a particular suitable average case the cost of the algorithms will be determined in terms of data transfers, comparisons and index comparisons. The total cost will be determined in terms of time. The time requirement has been

expressed using algebraic expressions. This expression can be easily minimized for a particular value of k , the degree of multi-valued architecture. To find out the optimal k numerical methods of finding roots of equation have been applied.

1.5 Organization of the Thesis

This thesis has been organized into the following chapters.

In Chapter 2 preliminary concepts on multi-valued architecture have been discussed. Multi-valued number system, memory storage requirement for multi-valued number system, concepts of multi-valued Logic gates and some important logic circuits have been discussed for this purpose. The concept of multi-valued ALU organization has also been discussed.

In Chapter 3 the binary concept introduced in the algorithm has been discussed. Divide and conquer approach and its complexity have been presented. The algorithms like binary search and quick sort which uses this concept have been discussed with their complexity analysis.

In Chapter 4 some multi-valued comparator circuits have been proposed. The proposed comparators are comparators using bit comparison, parallel comparators and pipelining comparators. These have been discussed mentioning their architectures and processing flow. The time requirements of these architectures for a comparison instruction have been roughly estimated in this chapter.

k -nary search algorithm, its complexity analysis for different architectures have been discussed in chapter 5. The optimal values of k for different architectures have been calculated in this chapter.

Chapter 6 is the most important chapter of thesis. An algorithm for quick-sort with k -partitions has been presented and the complexity analysis of the algorithm for different architectures has been presented. The optimal values of k for different size of problem space have been determined in this chapter. This chapter contains the achievements we have found in our research.

In chapter 7 some experimental results on quick-sort and binary search have been presented. The method of carrying out the experiment has also been described. Chapter 8 contains some suggestions for further improvement, further research plan and shortcomings of our research.

number systems. Octal numbers use digits from 0 to 7 and Hex numbers use digits from 0 to F. These two number systems have 8 and 16 digits respectively. They are nothing but application of binary number system in the sense that each of the 8 octal digits can be fully represented by 3 binary digits and each of the 16 Hex digits can be fully represented by 4 binary digits. Octal and Hex numbers are mainly used in expressing constants while writing programs.

Decimal numbers are the numbers, which is easily conceivable by humans. So this is very popular. It is very easy to add, subtract, multiply and divide. Floating point operations are very easily handled by using this type of numbers. So these numbers are used as a basis or intermediate format to transfer numbers from one number system to another.

2.3 k-nary Number System

k-nary number system is the number system where the radix is k and the possible digits of the number may be from 0 to (k-1). Theoretically k may be any number. A number in k-nary number system is defined by the following expression.

$$N = (b_{n-1}b_{n-2} \dots \dots \dots b_1b_0) = \sum_{i=0}^{n-1} k^i b_i$$

where $b_i = 0, 1, 2, \dots \dots \dots, (k - 1)$.

This is the generalized formula for multi-valued number system. This type of number system is not familiar. For example, if the value of $k=3$ then the system will be called as ternary number system. Ternary number system has some special property but this is not yet possible to implement because of non availability of suitable electronic circuits. So far no tristate stable device has been invented which may be suitable for ternary system. So we can theoretically develop systems using ternary logic but basically these are binary systems.

2.4 Memory Storage for k-nary Multi-valued Number System

Consider a memory organization which stores N numbers. The numbers to be saved will be in the range from 0 to M-1. In a k-nary number system to represent one number it requires $\log_k M$ digits. If the space required or complexity for one k-nary bit is C(k) then the total space required or complexity for N numbers can be shown by the following expression

$$S(k) = N \log_k MC(k)$$

considering $C(k) = kT$ where T is a constant we get $S(k) = NkT \frac{\log_e M}{\log_e k}$

Differentiating both the sides for optimal value of k we get the following relation

$$\begin{aligned}\frac{dS}{dk} &= \frac{d}{dk} \left(NkT \frac{\log_e M}{\log_e k} \right) \\ &= NT \log_e M \frac{d}{dk} \left(\frac{k}{\log_e k} \right) \\ &= NT \log_e M \frac{\log_e k - 1}{(\log_e k)^2}\end{aligned}$$

Now for the optimal value of k we get

$$\begin{aligned}\frac{dS}{dk} &= 0 \\ \Rightarrow NT \log_e M \frac{\log_e k - 1}{(\log_e k)^2} &= 0 \\ \Rightarrow \log_e k - 1 &= 0 \\ \Rightarrow k &= e\end{aligned}$$

So we can say that for $k = 2.71$ we hope to get an optimal number system. Putting $k = 2$ and $k = 3$ in the expression of S indicates that 3 is superior. Now we are going to find out the maximum space requirement for a ternary bit to get better performance than binary number system. From the equation of $S(k)$ we get the following equations.

$$S(2) = \log_2 M C(2) \quad S(3) = \log_3 M C(3)$$

$$\frac{S(3)}{S(2)} = \frac{\log_3 M C(3)}{\log_2 M C(2)} = \log_3 2 \frac{C(3)}{C(2)}$$

Now for $S(3) < S(2)$ we get

$$\log_3 2 \frac{C(3)}{C(2)} < 1$$

$$\Rightarrow C(3) < \log_2 3 C(2)$$

If the ternary device that may be invented follows the above mentioned inequality in VLSI design layout of RAM chip it will save our total space.

2.5 Binary Logic Gates

The basic binary gate used in digital electronics is NAND gate. The other necessary gates are AND, OR and NOT gates. NAND, AND and OR gates are all multiple (two or more) input gates with only one output. NOT gate or inverter has only one input and the output is just the inverted signal of input.

Following diagrams and tables show the block diagram of 2 input single output gates and single input single output gate with corresponding truth table. NAND gate is called the basic gate because all other gates can be easily formed from the NAND gate.

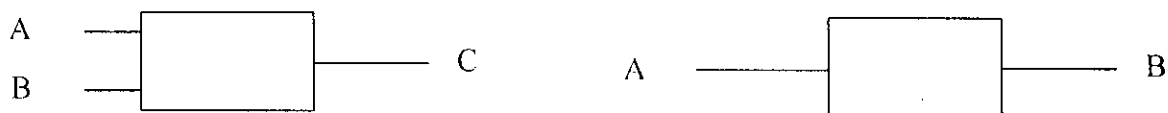


Figure 1 Block diagram of 2 input basic logic gates.

Table 1 Truth table for NAND gate.

A	B	C
0	0	1
0	1	1
1	0	1
1	1	0

Table 2 Truth table for AND gate.

A	B	C
0	0	0
0	1	0
1	0	0
1	1	1

Table 3 Truth table for OR gate.

A	B	C
0	0	0
0	1	1
1	0	1
1	1	1

Table 4 Truth table for NOT gate.

A	B
0	1
1	0

2.6 Concept of Multi-valued Logic Gates

The truth table of different logic gates will be changed with the introduction to multi-valued logic. In ternary logic the following truth table has been proposed for error detecting codes. The digits of the binary system has been changed to -1, 0 and 1.

Table 5 Truth table for ternary NOT gate.

A	B
-1(0V)	1
0(2.5V)	0
1(5V)	-1

The concept of AND, OR and NAND may be same as binary logic because this ternary operations can be easily derived from two consecutive binary operations. But the circuit must be modified with the expected change in ternary basic logic circuit or with the invention of ternary device in near future. Obviously the time requirement in ternary operations will be more demanding. But this extra time requirement will be compensated by the amount of computation done by ternary gates and will be contributory to the performance of ternary systems.

2.7 Important Binary Digital Logic Circuits

There are different logic circuits made by primitive logic gates. These circuits are very important in computer and processor design and specially in designing any sequential circuit. Such circuits are described below with block diagram.

Adder/Subtractor: Adder and subtractor are implemented using the same circuit. The block diagram of adder is shown below.

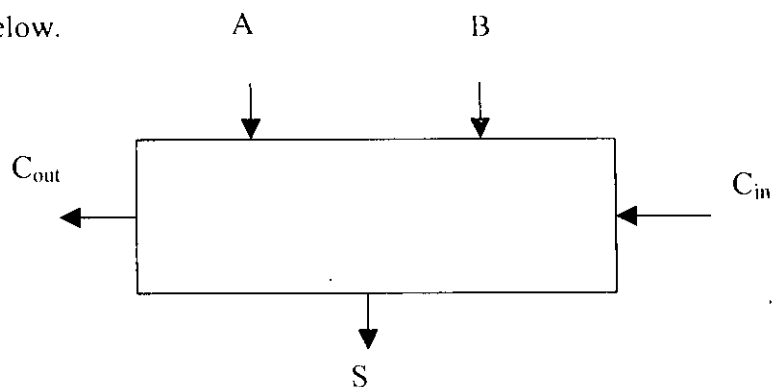


Figure 2 Block diagram of Traditional Adder/Subtractor

In the block diagram A and B are inputs with n bits. S is the output with n bits. C_{in} is the carry input and C_{out} carry output. While subtracting B from A the input B will be changed to inverted B. This will serve the purpose with little modification. This adder circuit is used as a comparator. When A is compared with B then subtraction is done at first then depending on the positive or negative result corresponding flag is set and the result of comparison is found in the flag register.

Registers: This is actually an array of D flip-flops. D flip-flop is nothing but an Electronic delay circuit. It has one input and one output. Some flip-flops supply inverted output also. The input reaches the output line until and unless a positive clock pulse is applied in the clock input of the D flip-flop. Some D flip-flops are leveled triggered. It means logical one input in the Enable pin allows the input to propagate to output. Register is an array of D flip-flops where the Clock inputs are tied together. The block diagram of the register is shown below.

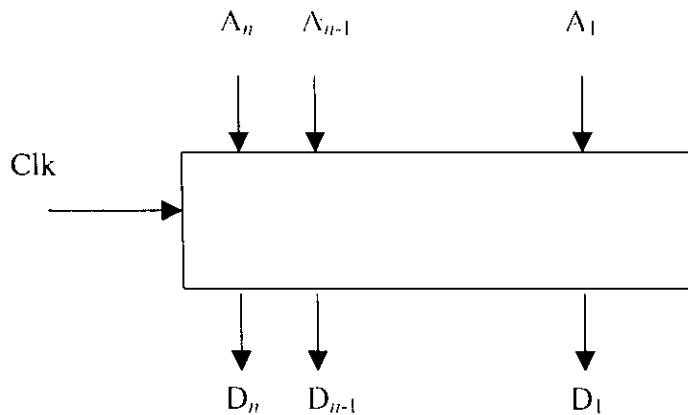


Figure 3 Block diagram of registers.

Counters: This is logically an array of T flip-flops which increments its value with each positive clock edge. Actually it has input lines for loading the initial value of the counter as many as output lines. The block diagram of counters are same as that of registers. Shift registers has the same principle of operation and the internal elements of the shift register are D flip-flops. It shifts the bits one position at a time in a particular direction specified by some input pins in the shift register. Counters are used in program counter and clock management and step detection circuits in the computer.

Decoder: This circuit decodes the binary numbers. This is formed by basic gates. The main purpose of the decoder circuit is shown in the next block diagram. There are n inputs in the circuit and 2^n outputs.

When a input is given only one pin at the output is activated which corresponds to the input number in the serial. It is used for coding purpose. This circuit is extensively used for decoding instructions.

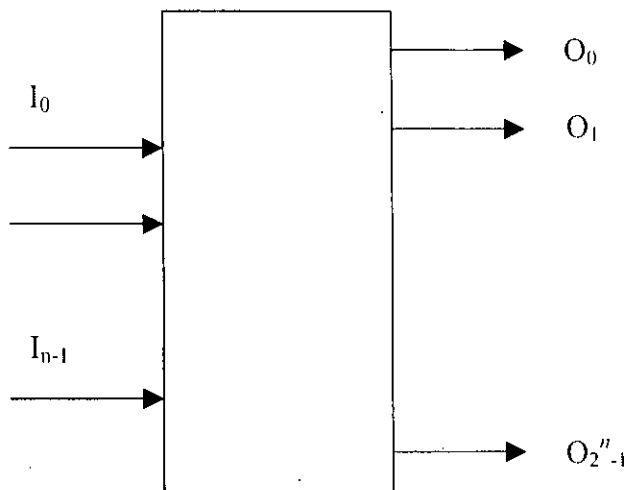


Figure 4 Block diagram of decoder.

2.8 Binary Architecture of ALU

ALU (Arithmetic Logic Unit) is one of the most important part of microprocessor. This section is responsible for computing operations of program i.e., addition, subtraction, shifting, multiplication, comparison, division etc. So speed of the computer mostly depends on the principles of computing ckt in ALU. Currently used conventional ALU are all capable of performing binary and unary operations. For example, it can add two numbers at a time. So this architecture is called binary architecture. This architecture is very simple in structure. The most important part of this ALU is a combinational circuit. This circuit performs different actions on different setting parameters in control inputs.

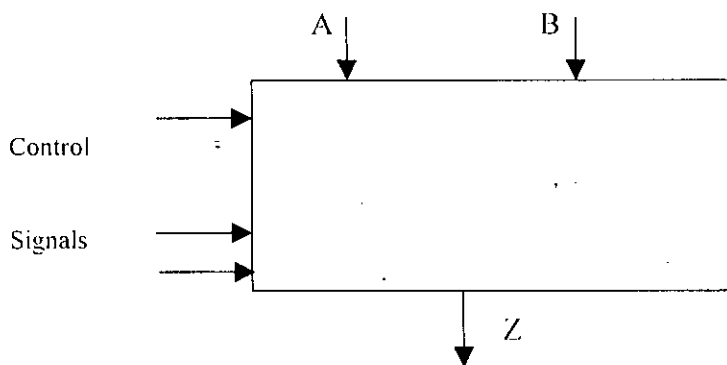


Figure 5 Block Diagram of binary ALU



The computing operations of ALU is done in one clock cycle and at the same time it saves the flag registers depending on the result found.

2.9 Multi-valued Architecture of ALU

The ALU which can process more than two operands at a time can be called multi-valued ALU. Then the architecture of the ALU will be changed a lot. There should be more input lines or more built in registers in ALU for data storing. This type of architecture must be capable of adding three or more operands at a time. The multiplier circuit may take the advantage of such an adder. The comparator circuit will compare one data with more than one data. This type of architecture will obviously be a very good one for programmers point of view. Shorter programs will be sufficient to do same task as the current binary architectures do in less time. To furnish the ALU with this feature the architecture of ALU must be changed. The ALU will be voluminous if we want to do the same work by a combinational circuit. Sequential circuit is an alternative to this combinational circuit. But it requires additional circuitry for time management. In sequential circuit ALU will work as a pseudo multi-valued processor. In every step it will perform a binary operation and such several steps will be required to perform like multi-valued system.

For multi-valued architecture of ALU the operations will be more complex. The Multi-valued addition and comparison are different. So by the same circuit this is impossible to perform simultaneously. Different modules will be designed for the different multi-valued operators. Thus a multi-valued architecture of ALU can be shown by the following block diagram.

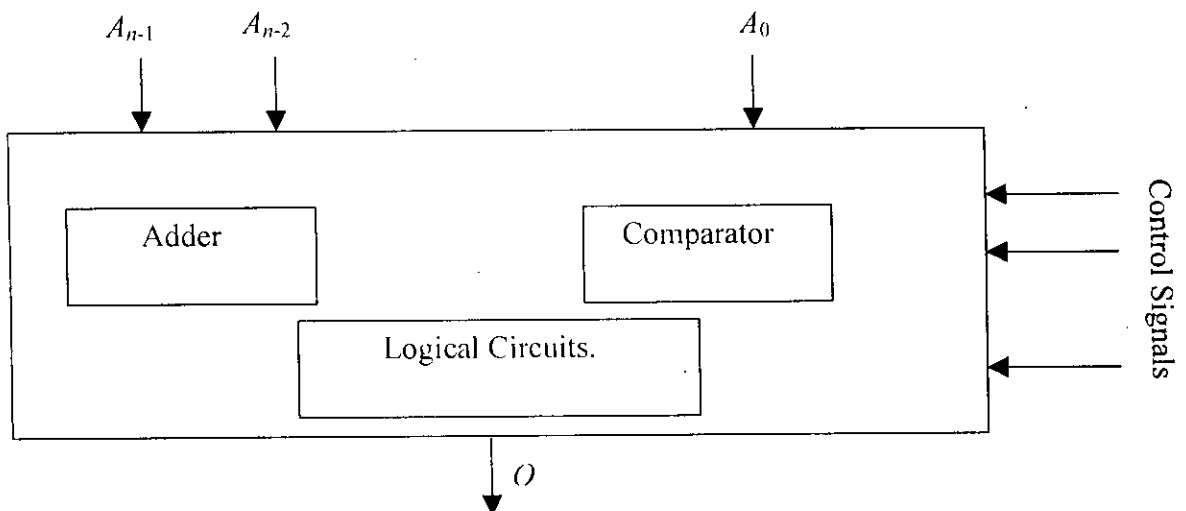


Figure 6 Conceptual block diagram of multi-valued ALU

Chapter 3

Algorithms using binary Concept

This chapter describes the preview of the algorithms that use binary concept. Divide and Conquer approach has been described with complexity analysis. The complexity analysis of binary search and quick-sort algorithm are also done here. This chapter also discusses the need for a new type of comparator that may be very efficient to improve the quality of binary search and quick-sort algorithm.

3.1 Divide and Conquer Approach of Problem Solving

This is actually a problem solving methodology in computer science. This approach itself an algorithm which can be applied to simplify some algorithms. The main principle of this algorithm is defined by the following steps.

Step 1: To split the problem space into several smaller problems.

Step 2: Solving the smaller problems separately.

Step 3: Merging the solution of smaller problems into the solution of the actual problem.

In step 2 of the algorithm to solve the smaller problem spaces divide and conquer approach can be further applied. So divide and conquer approach will be applied in recursive manner. In general sense Divide and Conquer approach is consistent with multi-valued approach. The total problem space may be divided into k parts. Traditional binary approach divides the problem space into two parts, i.e., $k=2$. The recursive algorithm for divide and conquer approach of problem solving is shown below.

Procedure Divide_and_Conquer(P , low, high)

P is the problem space and low and high are the indexes of bottom and top of the problem space.

Begin

 if low = high then return

 Divide the problem space into k spaces P_0, P_1, \dots, P_{k-1}

 for $i \leftarrow 0$ to $k-1$ do

 Divide_and_Conquer(P_i , low _{i} , high _{i})

 end for

 combine the small problem spaces P_0, P_1, \dots, P_{k-1} to generate the solved problem space P

end procedure.

To analyse the Divide and Conquer algorithm the following parameters have been defined.

$T(1)$ =Cost of solving a problem with only one element. This is actually the terminal condition.

N = Total number of elements in the problem.

$M(k, n)$ = Cost of merging k sub problems with n elements each.

The total cost can be expressed as follows.

$$\begin{aligned}
 T(N) &= kT\left(\frac{N}{k}\right) + M\left(k, \frac{N}{k}\right) \\
 &= k\left[kT\left(\frac{N}{k^2}\right) + M\left(k, \frac{N}{k^2}\right)\right] + M\left(k, \frac{N}{k}\right) \\
 &= k^l T\left(\frac{N}{k^l}\right) + k^{l-1} M\left(k, \frac{N}{k^l}\right) + k^{l-2} M\left(k, \frac{N}{k^{l-1}}\right) + \dots + M\left(k, \frac{N}{k}\right)
 \end{aligned}$$

$$N = k^l$$

So the cost function can be simplified to as follows.

$$\begin{aligned}
 T(N) &= NT(1) + \frac{N}{k} M(k, 1) + \frac{N}{k^2} M(k, k) + \frac{N}{k^3} M(k, k^2) + \dots + \frac{N}{k^l} M(k, k^{l-1}) \\
 &= NT(1) + N \sum_{i=0}^{l-1} \frac{1}{k^{i+1}} M(k, k^i) \\
 &= N \left[T(1) + \sum_{i=0}^{l-1} \frac{1}{k^{i+1}} M(k, k^i) \right]
 \end{aligned}$$

So the cost function is dependent on the size N of problem and the value of k . We can determine the optimal values of k for different algorithms. Actually it depends on the nature of the function M .

3.2 Algorithms using Divide and Conquer Approach

Divide and Conquer approach has been applied successfully in different algorithms. By applying this approach the performance of different algorithms can be improved substantially. Binary search algorithm uses this approach that can improve the performance from $O(n)$ to $O(\log_2 n)$. Quick sort applies the Divide and Conquer approach which improves the sorting cost from $O(n^2)$ to $O(n \log_2 n)$. Besides these, Divide and Conquer approach has been applied successfully in solving Systems of Linear Equations and the cost can be reduced by a constant factor. Strassen's matrix multiplication scheme uses the Divide and Conquer approach which can solve the problem with $O(n^{\log_2 7})$ cost rather than $O(n^3)$ cost. This approach can also be applied in closest pair problem by dividing the space into two parts in every recursive call of the algorithm. All these algorithms split the problem space into two parts, i.e., only binary approach has been applied in these algorithms. Recently some researchers have

found that ternary approach in Divide and Conquer approach is very much efficient in computational performance. So it is a matter of fact that exploration of algorithms using ternary Divide and Conquer approach may be prospective.

3.3 *Binary Search Algorithms*

Binary search is the searching algorithm that can find out an element from a sorted array by applying the Divide and Conquer approach. The main advantage of applying Divide and Conquer approach in searching is that the searching space becomes smaller logarithmically in every step and there is no need for combining search space. The algorithm is listed below.

Procedure BinSearch(A , low, high, v)

Begin

```
while (low < high) do
    mid  $\leftarrow$   $\lceil (low + high)/2 \rceil$ 
    if  $A[mid] > v$  then
        high  $\leftarrow$  mid
    else if  $A[mid] < v$  then
        low  $\leftarrow$  mid
    else return mid
```

end while

end procedure.

The main cost incurred here is the cost of comparison. The three comparisons in each iteration can be implemented by one COMPARE instruction of microprocessor. So in each step we need only one comparison. So the number of comparisons required in the worst case is $\log_2 n$, where n is the total number of elements in the array. To determine the average number of comparisons the following analysis can be done.

There is 1 element that can be found by only 1 comparison.

There are 2 elements that can be found by only 2 comparisons.

There are 2^2 elements that can be found by only 3 comparisons.

.....

There are 2^{l-1} elements that can be found by only l comparisons.

So average number of comparisons required for Binary search algorithm

$$= \frac{1 \times 1 + 2 \times 2 + 2^2 \times 3 + \dots + 2^{l-1} \times l}{n}$$
$$= \log_2 n - 1$$

3.4 Quick-sort Algorithm

The Divide and Conquer approach may be used to arrive at an efficient sorting method. This method makes two partitions of the array to be sorted. One partition contains elements less than a particular element of the array and the other partition contains elements larger or equal to that value. The Quick-sort algorithm is listed as follows.

Procedure Quick-sort(A, p, q)

A is the array to be sorted from the index p to q .

Begin

 if $p < q$ then

$j \leftarrow q + 1$

 partition(p, q, j)

 Quick-sort($A, p, j - 1$)

 Quick-sort($A, j + 1, q$)

 endif

end Procedure.

The partition procedure plays the important role in Quick-sort. So the details of partitioning is discussed below.

Procedure partition(m, p, q)

We are going to partitioning array A from index m to p . Currently $A(m)$ contains the partitioning element. q contains the index of partitioning element after partitioning. Thus the whole array will be partitioned by the value $A(m)$ and the array contains the values satisfying the following conditions.

$A(i) < A(m)$ for $0 \leq i \leq (q - 1)$

$A(i) > A(m)$ for $(q + 1) \leq i \leq p$

Begin

$v \leftarrow A(m)$ /*This is the partitioning element */

$i \leftarrow m$

do

do $i \leftarrow i + 1$ while $A(i) < v$

do $p \leftarrow p - 1$ while $A(p) > v$

if $i < p$ then

exchange $A(i)$ and $A(p)$

else

terminate $\leftarrow 1$

endif

while not terminate

$A(m) \leftarrow A(p)$

$A(p) \leftarrow v$

$q \leftarrow p$

end partition

3.5 Performance Analysis of Quick-sort Algorithm

For the simplicity of analysis we have considered a suitable average case. Let us consider $n=2^k$ and in every partitioning process the array is partitioned into two equal parts. While partitioning the array all the elements of the array will be compared with all other elements of the array. So if there is n elements in the array then total number of comparisons for partitioning is $n-1$. Thus the comparison cost can be defined as follows.

$$\begin{aligned} C(n) &= (n-1) + 2C\left(\frac{n}{2}\right) \\ &= (n-1) + 2 \times \left(\frac{n}{2} - 1\right) + 4C\left(\frac{n}{4}\right) \\ &= (n-1) + (n-2) + \dots + 2^{k-1} \left(\frac{n}{2^{k-1}} - 1\right) + 2^k C\left(\frac{n}{2^k}\right) \\ &= (n-1) + (n-2) + \dots + n - 2^{k-1} \\ &= kn - n = n(k-1) = n(\log_2 n - 1) \end{aligned}$$



Statistically in partitioning an element in one partition has the probability 0.5 to be transferred to the other partition. So the number of swapping can be defined as follows.

$$\begin{aligned}
 T(n) &= \frac{n}{4} + 2 \times T\left(\frac{n}{2}\right) \\
 &= \frac{n}{4} + \frac{n}{4} + 4T\left(\frac{n}{4}\right) \\
 &= \frac{n}{4} + \frac{n}{4} + \dots + \frac{n}{4} + 2^k T\left(\frac{n}{2^k}\right) \\
 &= \frac{kn}{4}
 \end{aligned}$$

So total number of data transfer operations $= \frac{3}{4} n \log_2 n$

3.6 Need for Multi-valued Architectures

Binary search and Quick-sort algorithm need comparison extensively and comparison plays contributory role in the cost function. If we use ternary search algorithm or Quick-sort with more than two partitions then it is necessary to compare one element with two or more elements. But our conventional comparator circuit in ALU does not support this provision. So we have to write more than one comparison instruction for such multi-valued comparison. To execute such instructions it requires overheads for loading the instructions. So it requires more time. To get better efficiency we can change the comparator circuit or add a new comparator unit in the ALU. Such type of unit should be able to load the data and compare the elements to give the result by one instruction. Hopefully such multi-valued comparator circuit will be able to improve the performance of Quick-sort and searching algorithms because these algorithms use this type of ready made instructions. In the following chapters such multi-valued comparators will be discussed and the effect on the performance of algorithms will be also analysed.

Chapter 4

Proposed Multi-valued Comparator Circuits

In this chapter the concept of multi-valued comparator has been presented. Three architectures for multi-valued comparator have been proposed in this chapter. These are comparator by bit comparison, parallel comparator and pipelining comparator. The block diagrams and processing flow of comparison have been discussed in details for each proposed comparator blocks. From this discussion some assumption have been made to determine the time requirement of one COMPARE instruction. With the same assumption time requirement by COMPARE instruction by binary comparator and new proposed multi-valued comparators have been estimated in terms of T units of time where T denotes the time requirement to load a data into a register from a physical memory location.

4.1 Binary comparison

Consider an element A . We want to compare this element with a value v . Then the function of comparison is defined as follows

if $v \leq A$ then $\text{compare}(A, v) = 0$

if $v > A$ then $\text{compare}(A, v) = 1$.

The circuit which follows the above mentioned criteria is called binary comparator.

4.2 Multi-valued Comparison

Now consider an array A with more than one element. The $(k-1)$ elements of the array with $A[0] \leq A[1] \leq A[2] \leq \dots \leq A[k-2]$ are sorted. The value v will be compared with the elements of the array. The comparison results may be described as follows.

if $v \leq A[0]$ then $\text{compare}(A, v) = 0$

if $A[0] < v \leq A[1]$ then $\text{compare}(A, v) = 1$

if $A[1] < v \leq A[2]$ then $\text{compare}(A, v) = 2$

.....

.....

if $A[k-3] < v \leq A[k-2]$ then $\text{compare}(A, v) = k-2$

if $v > A[k-2]$ then $\text{compare}(A, v) = k-1$

The circuit which follows the above mentioned criteria is called the k -valued comparator. If the value of k is 3, i.e., we are going to compare one element with two other sorted elements then the comparator circuit is called ternary comparator and the comparison is called ternary comparison. The result of ternary comparison will be 0, 1 or 2. The result of k -valued comparison will be k integers from 0 to $(k - 1)$ and there are $(k - 1)$ elements to be compared with.

4.3 Current Comparator Circuits used in Computers

The following block diagram represents a typical n bit comparator circuit which may be used in the ALU of the computer or equivalent to the Adder/Subtractor unit of ALU.

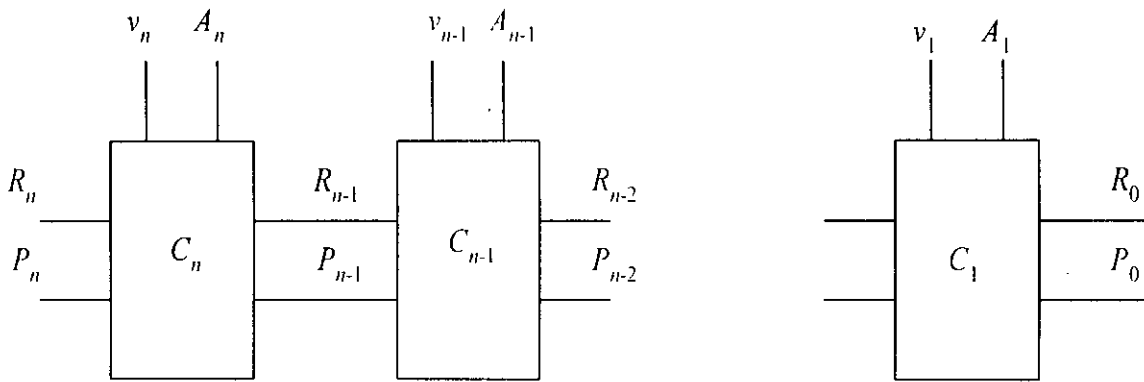


Figure 7 Detailed Block diagram of comparator.

The result of the comparison will be found on R_0 . The output will follow the following law

if $v > A$ then $R_0 = 1$

if $v < A$ then $R_0 = 0$

P_i is the output which states whether the value of comparison already determined or not. If P_i is 1 somewhere only then the value of R_i will be propagated to the next portion. $P_{i-1}=0$ means R_{i-1} will be determined depending on the value of v_i and A_i . The truth table of the circuit is shown below

Table 6 Truth table of basic block of Comparator.

v_i	A_i	R'_{i+1}	P'_{i+1}
0	0	×	0
0	1	0	1
1	0	1	1
1	1	×	0

P'_{i+1} and R'_{i+1} are two intermediate result steps. the Boolean expression of the these terms are shown below. The value of P_0 will show whether the elements are equal or not.

$$R'_{i+1} = \bar{A}_i$$

$$P'_{i+1} = v_i \oplus A_i$$

The truth table of the final output is shown below

Table 7 Truth table of final output of Comparator.

P_i	R_{i+1}	P_{i+1}
0	R'_{i+1}	P'_{i+1}
1	R_i	P_i

The circuit diagram to implement the basic blocks is shown in the following figure.

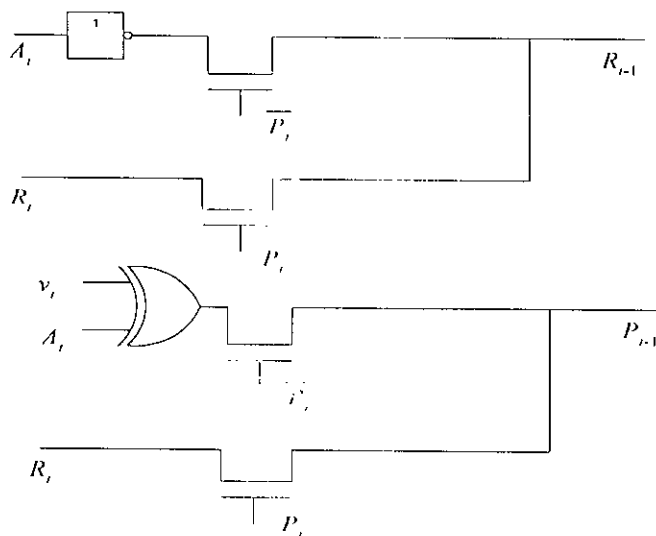


Figure 8 Circuit diagram of basic block of currently used comparators.

4.4 Proposed Multi-valued Comparator Circuits

The concept of comparator for binary comparison cannot be applied for the ternary comparators. Because by comparing only one bit it is quite impossible to determine the position where v can be inserted. There may be some alternatives to do this. These are listed below.

- i) Multi-valued comparison can be done by combinational circuits but it will be very voluminous and non modular. So it may not be feasible
- ii) For multi-valued comparison modular architecture with sequential circuit can be designed.
- iii) Parallel architecture is very much suitable for such structure.
- iv) A pipelining architecture can solve the problem like a sequential circuit.

4.5 Comparator by Bit comparison

Let there be $(k-1)$ sorted values $A^{(0)}, A^{(1)}, A^{(2)}, A^{(3)}, \dots, A^{(k-2)}$ to be compared with one value v . Each values are of length n bits. This method finds the result of comparison by applying the following algorithm.

- i) Let $v_i, A_i^{(j)}$ be the i th bit of v and $A^{(j)}$. For each bit i Compare v_i with $A_i^{(0)}, A_i^{(1)}, A_i^{(2)}, A_i^{(3)}, \dots$ to find the index j such that $v_i = A_i^{(j)}$ for lowest value of j .
- ii) Go to the next bit comparison if such index is found and start comparison step (i) from j .
- iii) It may be found that there is no value in the array from j to $k-2$ which follows the above mentioned conditions in step (i). Then the search terminates deciding the following results of the comparison.
if $v_i < A_i^{(j)}$ then the result is $(j-1)$.
if $v_i \geq A_i^{(j)}$ then the result is j .

4.5.1 H/W to implement this type of comparator

The block diagram of the comparator performing the above mentioned operations is shown in the next page. Here we are describing the different parts of the comparator unit.

4.5.2 Comparator of bits between v_i and $A_i^{(j)}$

The inputs of this block are v_i and $A_i^{(j)}$. There are three outputs:-

- (i) OF (Output Found), i.e, the circuit has found a position in the array for value v .
- (ii) RIO (Row Increment Order). This output shows that the bits $A_i^{(j)}$ is lower than v_i , so the position of the insertion in the array must increase by one.
- (iii) BIO (Bit Increment Order). This output shows that $v_i = A_i^{(j)}$, so searching in this row is valueless and the next bit should be compared.

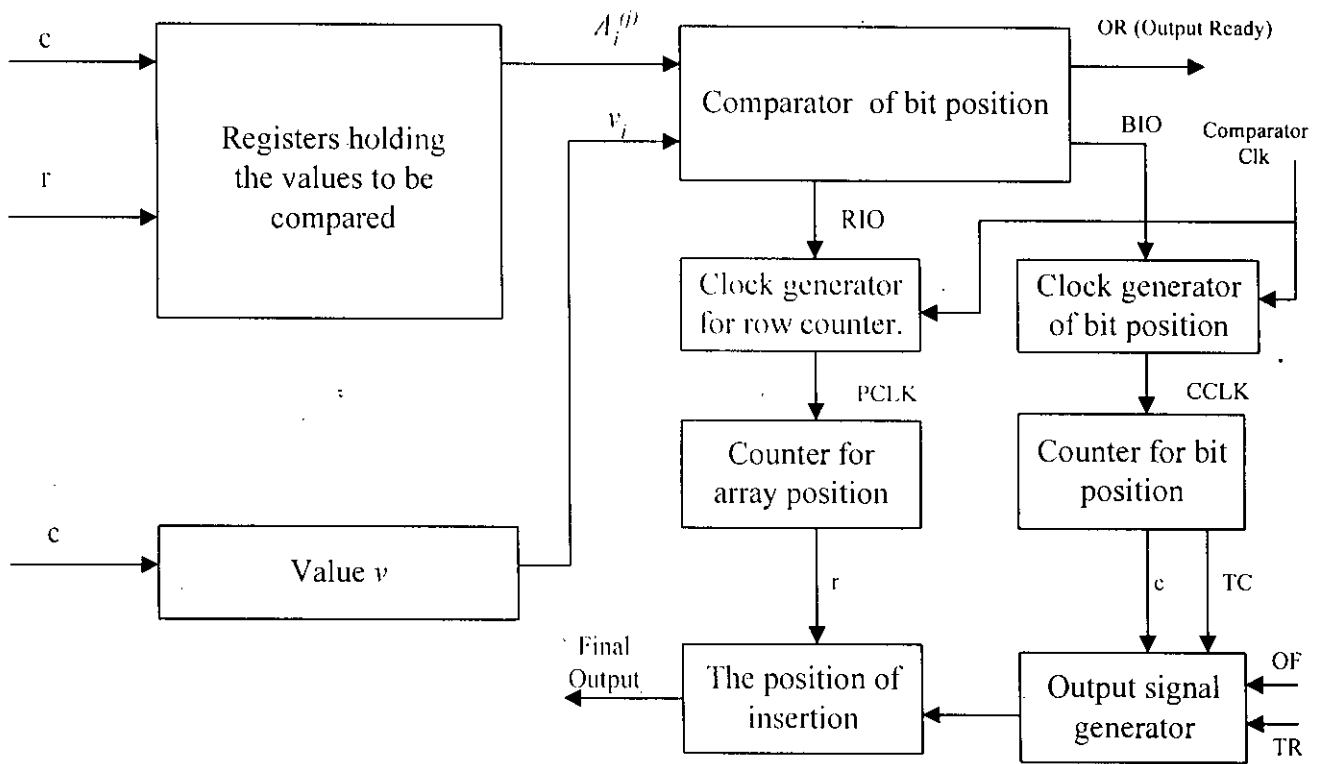


Figure 9 Block Diagram of the proposed comparator unit by bit comparison.

The truth table of the above mentioned circuit is given as follows.

Table 8 Truth table for finding OF, RIO,BIO

v_i	$A_i^{(j)}$	OF	RIO	BIO
0	0	0	0	1
0	1	0	1	0
1	0	1	0	0
1	1	0	0	1

The logic equations of the OF, RIO, BIO output is given below.

$$\text{OF} = v_i \overline{A_i^{(j)}}$$

$$\text{RIO} = \overline{v_i} A_i^{(j)}$$

$$\text{BIO} = \overline{v_i} \overline{A_i^{(j)}} + v_i A_i^{(j)}$$

4.5.3 Position counter

This is a normally edge triggered counter. There must have a clock input. The clock **Clk** is the clock of comparator. The width of the clock must be sufficient to do all the work of the comparator circuit. The minimum width of clock can be discussed after describing all the portions and functions of this comparator.

This clock of this counter gives zero to high transition whenever the circuit begins to compare with new element. Actually this is not the regular clock of the comparator. The generation of this clock is determined by

$$\text{PClk} = \text{RIO.Clk}$$

The output of the position counter is **r** which actually shows which row of the array is going to be searched. This can be easily implemented by recognizing the output bits of the counter. Terminal condition of the counter can be found by the output TR(Terminal Row). $\text{TR} = R_3 R_2 R_1 R_0$, if the comparator is 16 valued comparator. If the comparator is 10 valued then

$$\text{TR} = R_3 \overline{R_2} R_1 \overline{R_0}$$

The number of bits in the counter depends on the degree of comparison. If the comparator is k valued the number of bits in the position counter is $\lceil \log_2 k \rceil$.

4.5.4 Bit Counter

The input of bit counter is a pulse. This pulse goes low to high when next bit is going to be searched instead of current bit. The pulse is generated by another circuit of which logic equation is shown below.

$CClk = BIO.Clk$, where Clk is the clock of comparator.

The output of the counter is the value of bit position which is going to be searched currently. The number of bits required is denoted by $\lceil \log_2 n \rceil$, where n is the number of bits of elements to be compared. Another important output is TC (Terminal Column). This is determined by recognizing the terminal bit position indicator in the counter. This can be determined by the same way as that of TR. Both Bit and Position counter must have a clear control signal input to reset the counters at the beginning of comparison.

4.5.5 Output Available Signal

The output of the comparator is the position of the array where the element v can be inserted in the sorted array. Actually this position number is stored in the counter at the time of comparison. The time when output is available is determined by the following events.

- (i) Terminal Row has been reached.
- (ii) Terminal Column has been reached and v_i and $A_i^{(j)}$ are equal.
- (iii) OF is high or On.

Combining all the events the following logic equation can be derived

$$OR'(Output\ Ready) = TR + TC \cdot \overline{OF} + OF$$

To make the OR available in the next half of the clock cycle it is passed through edge triggered flip flop by the following way :-

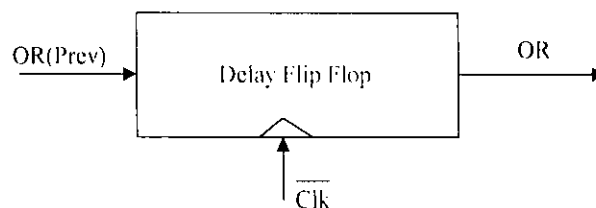


Figure 10 Block diagram of Output Ready signal generator.

This OR signal is used as a strobe signal of level triggered latch. We can say that the output of the circuit will be available at the output buffer for insert position in the array when OR signal is On and Clk signal is Off. That is when $OR \cdot Clk'$ is high it shows the operation of comparison has come to an end.

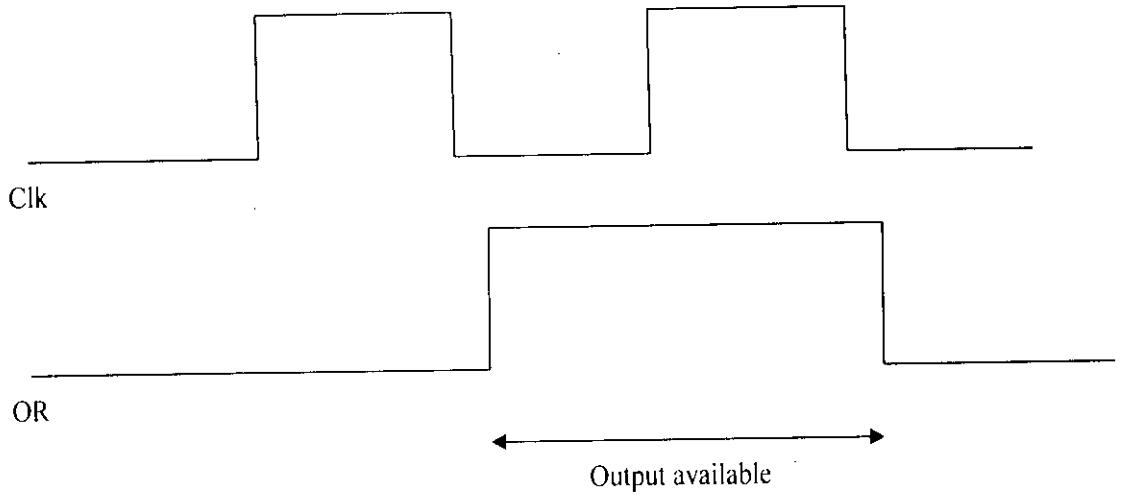


Figure 11 Timing diagram of Clock and Output Ready signal.

4.5.6 Registers Holding the array

There must be as many registers as the degree of comparisons. Each register is a shift register. The

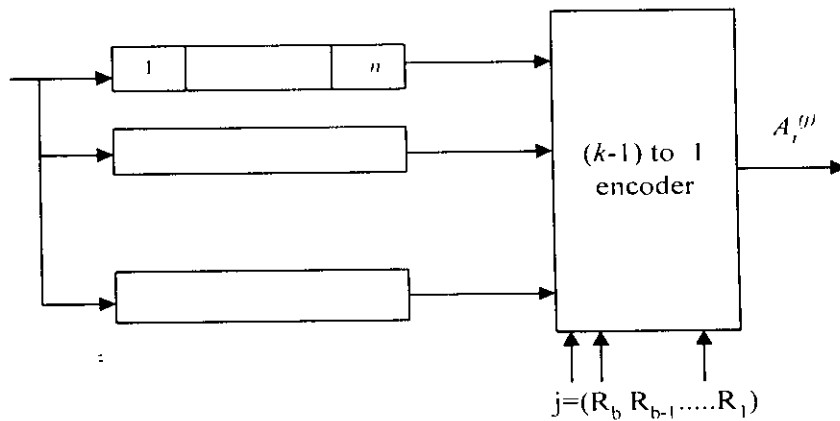


Figure 12 Block diagram of Register Array

registers shift their value one bit left in each low to high clock edge. So the clock input of the registers may be BClk used in Bit position counter and generated from the main comparator circuit. An encoder may be placed at the end of the register array. The serial output of each register will be connected to the encoder. The output of position counter i.e., $(R_{b-1}, R_{b-2}, \dots, R_0)$ will be connected to the input of encoder. The block and symbolic diagram of register array is shown in the Figure 12. For taking input to the register there should be some connection between register input to the microprocessor bus. Each register must have an enable input pin which allows to enter data from system bus to the register. Separate instructions can be employed to load data to the registers.

4.5.7 Value Register

This is the shift register which contains the value to be inserted. The register can be loaded like the register array. Only one clock input (BClk) is sufficient to use this register. The block diagram of this register is the same as the array registers discussed in the previous subsection.

The following components will be necessary for the k -valued comparator.

- (i) n bit shift register k units.
- (ii) $k-1$ to 1 encoder.
- (iii) $b = \lceil \log_2 k \rceil$ bit position counter and latch.
- (iv) $\lceil \log_2 n \rceil$ bit counter to determine which bit is going to be compared now.

The following are fixed H/W for comparator unit of any degree.

- (v) Comparator circuit
- (vi) Logic Circuit for OR, PClk and BClk.
- (vii) Flip flop to hold the value of OR.

4.6 Parallelism in Comparison

In parallel architecture the $(k-1)$ operands of the k -valued comparison will be compared with the

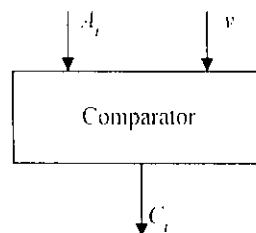


Figure 13 Single unit comparator

value v simultaneously by $(k-1)$ traditional binary comparators. The result of each comparison is one bit binary value. Such a binary comparator is described below.

$$C_i = 1, \text{ if } v \leq A_i,$$

$$C_i = 0, \text{ if } v > A_i,$$

The output of comparator C_i s are then fed to a combinational circuit. This combinational ckt determines the position where the value v can be inserted in the sorted list A . The organization of the comparator can be shown in the following figure.

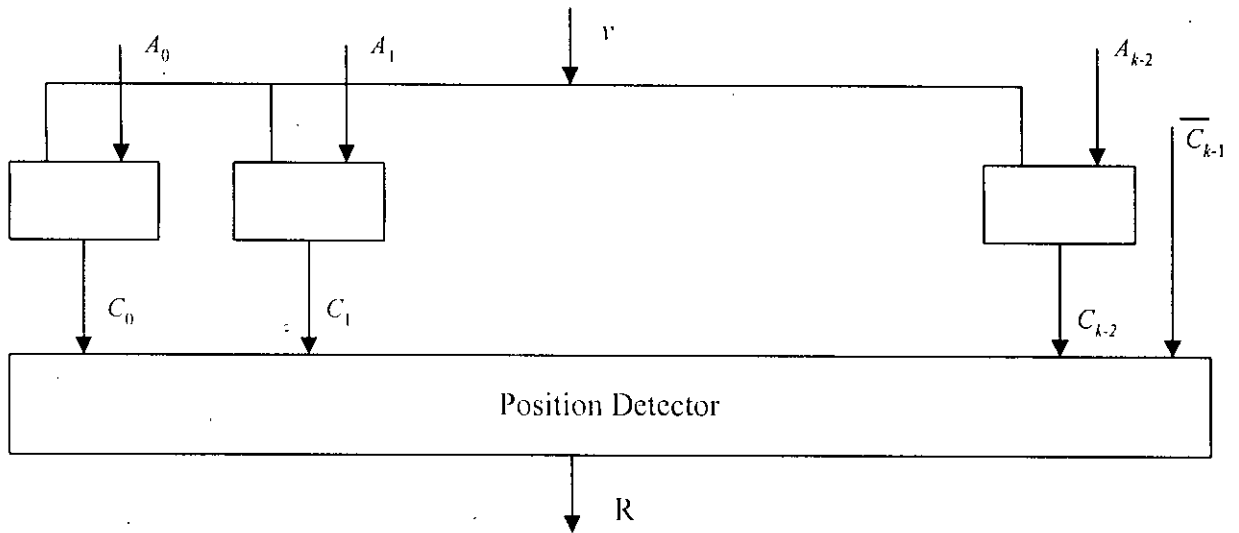


Figure 14 Block diagram of parallel comparator

R is the value from 0 to $(k-1)$ detecting position where the value is going to be inserted. It requires $\lceil \log_2 k \rceil$ bits to represent R . The position detector circuit can be implemented by the following way.

The circuit PD_i works according to the following principle

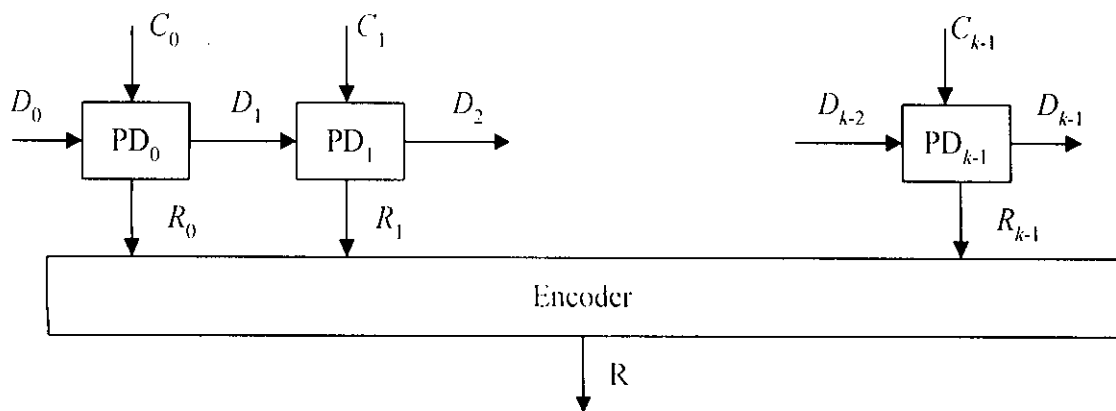


Figure 15 Block diagram of position detector

If D_{i-1} is 0 then it does indicate the position is yet to detect. So R_i and D_i would be C_i . If D_{i-1} is 1 then no need to detect position as it has already detected and $A_i = 0$.

Thus The equations R_i and D_i can be expressed as follows.

$$R_i = \overline{D_{i-1}} C_i$$

$$D_i = \overline{D_{i-1}} C_i + D_{i-1}$$

The last input to the encoder is C'_{k-1} that is the value v is greater than all the values of sorted array A .

4.7 Comparator Using Pipelining Idea

We know that modern processors use pipelining in different stages. For example, in instruction execution different phases can be pipelined together. Operand fetching, memory addressing, arithmetic unit processing and storing result can be pipelined. This method speeds up the processing substantially. Generally any repetitive job can be pipelined by dividing the job into some phases. For example if there are p phases of a process where each phase needs input from previous phase then the block diagram of pipelined process is shown in the next page.

Now if the process runs n times then the required time is defined by

$$\left(1 + \frac{n-1}{p}\right)T$$

Where T is the time required to run a single process without pipelining.

The main function to design a pipelined processor is to divide the phases such that it can be done separately. To get optimal output, the phases should be equal in time requirement. If highest time requirement in any one of the phases is T_m other than (T/p) then total time requirement to run the n processes is defined by

$$T + T_m(n-1)$$

and the first process requires the full time.

Multi-valued comparison can be easily done by using pipelining idea. We can assume multi-valued comparison as a repetition of single valued comparison. The algorithm is given below:-

Step 1: load the value v ,

Set the value of $i \leftarrow 0$

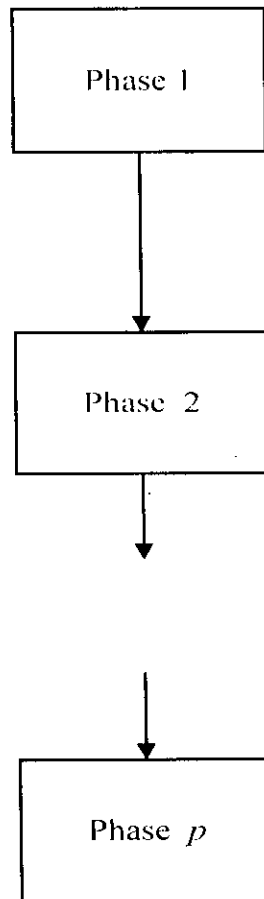


Figure 16 Pipe line Procesor

Step 2: load the element $A[i]$ from memory.

Step 3: compare the value v with $A[i]$

Step 4: Incrementing a counter i

Step 5: If the result of comparison is like $v \geq A[i]$ or $i \geq k$ then no need to execute the iteration. Otherwise, go to Step 2.

Step 2 to Step 5 of the algorithm work in a repetitive fashion. So these steps can be pipelined easily. Careful study shows that the phases and components to implement that phase are listed below.

Phase 1

This phase is responsible for calculating the address of data in the sorted array. The address of the operands will be stored in the instruction and this phase will do the following operations:-

- i) Fetching operand address from memory
- ii) Determining the physical address of memory by MMU.

Phase 2

This phase puts the physical address to the main memory or cache and waits for data, after read operations. Generally it takes same time to retrieve a byte, word, long word, floating point and double precision numbers. The reason behind it is that 64 bits are retrieved at a time from memory.

In phase 1 more than one addresses can be retrieved from the instruction. So it is not necessary to access memory in each iteration. Determining the physical address by MMU takes less time than data retrieval. So Phase 1 and Phase 2 take identical time.

Phase 3

This phase is responsible for arithmetic operation. This actually needs a subtraction operation. The time requirement seems to be same as phase 1 and phase 2. This phase needs logic circuits to compare two values.

Phase 4

The operations carried out by this phase are

- Increasing a counter by 1
- Taking decisions from the value of flags whether the processor has already found the final result.
- If the result is found then sending the index to accumulator or any other register of the processor.

This phase requires less time than previously mentioned phases.

4.8 Time Requirement in three proposed Architecture for comparison

The proposed three architectures of the comparators require different amount of time to compare data. In the following subsections the cost (time requirement) of different architectures will be discussed.

4.8.1 Time Requirement by Traditional Binary Comparison

The time requirement for binary comparison can be divided into two steps.

- (i) Time requirement to load the operand.
- (ii) Time requirement to compare the bits.

To load the operand in the ALU it requires three steps to be performed. These are

- (a) Fetching address and instruction.
- (b) Physical address calculation by MMU.
- (c) Finally memory content will be transferred to the ALU.

To perform these three operations we can assume that it requires T units of time for each steps on the average. On the other hand comparison circuit requires $T/2$ time units because it requires $n/2$ comparisons on the average to perform the total comparison. So time required by the compare instruction is $7T/2$.

4.8.2 Time requirement of Comparator by doing Bit comparison

The time requirement for k -valued comparison can be divided into two parts.

- (i) Time requirement to load $(k-1)$ values in register.
- (ii) Time requirement to compare the bit positions by the circuit.

Loading $(k - 1)$ values from memory requires a very high time in comparison to second part and main part of the comparison. So we suggest a pipelining structure for loading values from memory. It may have three parts. These are listed below:

Step 1: This step includes fetching the address of memory addresses from the instruction words stored in memory. At a time two or more addresses can be retrieved from memory.

Step 2: In this step the physical addresses of the memory contents of the array is calculated from their logical addresses. This operation is done by MMU (Memory Management Unit).

Step 3: In this step the physical address is supplied to the memory and the data is retrieved from the memory. The data is stored in the register.

All the above mentioned operations require about equal time. Let, the time requirement in any step take T time units.

So, to load $(k-1)$ values of stored array the time requirement is shown as follows:

= Time requirement to load first value of the array + Time requirement to load next $(k-2)$ values of the array

$$= 3T + (k-2)T$$

$$= kT + T$$

$$= (k+1)T$$

We can assume the time taken to determine the position of insertion in the array as T units. This includes

- i) Comparing $n/2$ bits on the average
- ii) Increasing the position counter $(k-1)/2$ times on the average
- iii) Increasing the bit counter $n/2$ times on the average

We hope that to do these operations it requires at best T units of time.

So total time requirement for k -valued compare instruction = $(k+2)T$

4.8.3 Time requirement of Parallel Comparator

The time requirement for k -valued comparison can be divided into three parts.

- i) Time requirement to load $(k-1)$ values in the register.
- ii) Time requirement to compare each element of the array with the value in each comparators in parallel.
- iii) Time requirement in position detection.

We can use the similar pipelining strategy to load the data from memory to registers of the comparator as described in the comparator circuit by bit comparing. It actually requires three pipelining steps. All the steps require no more than T units of time to complete their job. So, time requirement by loading $(k-1)$ operands is given by following expression.

$$3T + (k-2)T = (k+1)T$$

Comparing one value with other i.e., binary comparison require fixed T units of time considering all the bits are compared to get result. Position decoder requires at best T units of time. These two steps require same time because the internal circuits are same. So total time requirement in k valued comparison is $(k + 3)T$.

4.8.4 Time requirement for Comparators with pipelining

Observing the microprocessor Data Hand book we have found that following are the comparative time requirement in different phases

Table 9 Time requirement in different steps of pipelined comparator.

Phase number	Time req.
Phase 1	T
Phase 2	T
Phase 3	T
Phase 4	$T/2$

We hope that the cost of k -valued compare instruction will be equal to comparing $(k - 1)/2$ elements on the average. So, roughly the cost requirement is defined by-

$$4T \text{ (for comparing first element)} + (k - 1) \times T/2 \text{ (For comparing other elements)}$$

$$= kT/2 + 7T/2 = (k + 7)T/2$$

Chapter 5

Multi-valued k -nary Search

In this chapter k -nary search algorithm has been presented. The cost requirements of k -nary search algorithm by using binary architecture and new proposed architectures have been analysed here. The cost function have been minimized for optimal values of k . The discussion shows that none of the multi-valued architectures for comparator is efficient for k -nary search algorithm. Binary search algorithm is the best in the worst case even after introducing multi-valued comparator. The chapter concludes with a table comparing the time requirement by different comparators for k -nary search algorithm.

5.1 Algorithm of multi-valued k -nary Search

k -nary search algorithm can be easily written by slightly modifying the binary search algorithm. The search space is divided into k different parts in every iterations of the algorithm. In every iteration the value is compared with $(k - 1)$ boundary values of the partition segments and then the appropriate partition segment is chosen. The modified binary search algorithm is listed below.

Procedure k -narySearch(A , low , $high$, v , k)

A is the array to be searched. low and $high$ are the indexes of lower and upper boundary. v is the value to be searched. k is the degree of multi-valued comparison.

Array $partition[k]$

Begin

while ($low < high$) do

if $high - low + 1 < k$ then

for $i \leftarrow low$ to $high$ by 1 do $partition[i - low] \leftarrow A[i]$

return compare($partition$, k , v)

endif

$partition_size \leftarrow \frac{high - low + 1}{k}$

for $i \leftarrow 0$ to $(k - 2)$ do

$partition[i] \leftarrow low + partition_size \times i$

end for

$partition[k - 1] \leftarrow high$

$j \leftarrow compare(partition, k, v)$

if $A[j] = v$ then return the index.

else

$low \leftarrow partition[j]$
 $high \leftarrow partition[j + 1] - 1$

endif

end while

End Procedure.

5.2 Cost Requirement using Binary Comparator

Ignoring the cost of index comparison or calculation the main cost incurred here is the cost of element comparison. In every step of the iteration it requires $(k - 1)$ comparison in the worst case and $(k+1)/2 - 1/k$ comparisons in the average case for $k > 2$. So the total cost can be expressed by the following formula.

$$\begin{aligned}
 C(n, k) &= C\left(\frac{n}{k}, k\right) + \frac{k+1}{2} - \frac{1}{k} \\
 &= C\left(\frac{n}{k^2}, k\right) + \frac{k+1}{2} - \frac{1}{k} + \frac{k+1}{2} - \frac{1}{k} \\
 &= C\left(\frac{n}{k^l}, k\right) + l \times \frac{k-1}{2} - l \times \frac{1}{k} \\
 &= C(1, k) + l \times \frac{k+1}{2} - l \times \frac{1}{k} \text{ as } n = k^l \\
 &= \left(\frac{k+1}{2} - \frac{1}{k}\right) \log_k n, \text{ when } k \geq 3
 \end{aligned}$$

But when $k=2$ then in every iteration it requires one comparison so the cost of comparison is defined by $\log_2 n$. From the two type of cost it can be easily shown that

$$\frac{C(n, 2)}{C(n, 3)} = \frac{3 \log_2 n}{5 \log_3 n} = 0.6 \log_2 3 < 1$$

So Binary Search algorithm is better than ternary search algorithm for conventional comparator circuit. Now we are going to find out the value of k for which the cost is minimum for $k \geq 3$. To do this we have to differentiate the function $C(n, k)$.

$$\begin{aligned}
 \frac{dC}{dk} &= \frac{d}{dk} \left\{ \left(\frac{k+1}{2} - \frac{1}{k} \right) \log_k n \right\} = 0 \\
 \Rightarrow \frac{d}{dk} \left\{ \left(\frac{k+1}{2} - \frac{1}{k} \right) \frac{\log_e n}{\log_e k} \right\} &= 0
 \end{aligned}$$

$$\Rightarrow \log_e n \frac{d}{dk} \left\{ \frac{\frac{k+1}{2} - \frac{1}{k}}{\log_e k} \right\} = 0$$

$$\Rightarrow \frac{\log_e k \left(\frac{1}{2} + \frac{1}{k^2} \right) - \frac{1}{k} \left(\frac{k+1}{2} - \frac{1}{k} \right)}{(\log_e k)^2} = 0 \Rightarrow \log_e k \left(\frac{1}{2} + \frac{1}{k^2} \right) - \frac{1}{k} \left(\frac{k+1}{2} - \frac{1}{k} \right) = 0$$

We can get the optimal value of k solving this equation. But unfortunately for $k \geq 3$ there is no values of k which satisfies the equation. It means the cost function is ever increasing from $k=3$. It shows the optimal value of k is 3. It can also be shown by observing the $\frac{C'(n, k+1)}{C'(n, k)}$ term.

5.3 Cost Requirement using Comparators by Bit comparison

Time time requirement for a single multi-valued comparison instruction is $(k+2)T$ as shown in section 4.5. Now the cost function will be changed as follows

$$C(n, k) = C\left(\frac{n}{k}, k\right) + (k+2)T$$

$$= C\left(\frac{n}{k^2}, k\right) + (k+2)T + (k+2)T$$

$$= C\left(\frac{n}{k^l}, k\right) + l \times (k+2)T$$

$$= C(1, k) + l \times (k+2)T \text{ as } n = k^l$$

$$= (k+2)T \log_k n$$

Similarly differentiating both the sides of the cost function we get

$$\frac{dC}{dk} = \frac{d}{dk} \{(k+2)T \log_k n\} = 0$$

$$\Rightarrow \frac{d}{dk} \left\{ (k+2)T \frac{\log_e n}{\log_e k} \right\} = 0$$

$$\Rightarrow T \log_e n \frac{d}{dk} \left\{ \frac{k+2}{\log_e k} \right\} = 0$$

$$\Rightarrow \frac{\log_e k - \frac{k+2}{k}}{(\log_e k)^2} = 0$$

$$\Rightarrow \log_e k - 1 - \frac{2}{k} = 0$$

Applying the bisection method of solving roots of equation we have found that the roots of equation lies between $k = 4$ and $k = 5$. The exact root is $k = 4.32$. Putting the value $k = 4$ and $k = 5$ in $C(n, k)$ it shows that $C(n, 4)$ is lower. So for comparator by bit comparison processor the optimal degree of multi-valued comparison is $k = 4$.

5.4 Cost Requirement using Parallel comparator

As the time requirement for one multi-valued comparison instruction is $(k + 3)T$ as shown in 4.8.3 we can determine the cost function for k -nary search using parallel comparator is given below:

$$C(k, n) = (k + 3)T \log_k n \text{ unit time.}$$

Now differentiating both the sides we get the following function of k .

$$\log_e k - 1 - \frac{3}{k} = 0$$

Applying the bisection method of solving roots of equation we have found that the roots of equation lies between $k = 4$ and $k = 5$. The exact root is $k = 4.97$. Putting $k = 4$ and 5 in the cost function we get for parallel comparator the optimal degree of multi-valued comparison is $k = 5$.

5.5 Cost Requirement using pipelined comparator

As the time requirement for one multi-valued comparison instruction is $(k + 7)T/2$ as shown in 4.8.4 we can determine the cost function for k -nary search using parallel comparator is given below:

$$C(k, n) = \left(\frac{k + 7}{2}\right)T \log_k n \text{ unit time.}$$

Now differentiating both the sides we get the following function of k .

$$\log_e k - 1 - \frac{7}{k} = 0$$

Applying the bisection method of solving roots of equation we have found that the roots of equation lies between $k = 7$ and $k = 8$. The exact root is $k = 7.19$. Putting $k = 7$ and $k = 8$ in cost function we get for comparator by pipelining processor the optimal degree of multi-valued comparison is $k=7$.

5.6 Comparison among the optimal performance of different comparators for k -nary searching algorithm

The following table shows the approximate time requirement for searching algorithm by different type of comparators.

Table 10 Complexity of k -nary search for different multi-valued comparator

Type of Comparator	Time required for one comparison	Optimal value of k	Time required for searching in an array of n elements
Binary	$7T/2$	3	$5.83T \log_3 n$
Bit Comparison	$(k+2)T$	4	$6T \log_4 n$
Parallel comparator	$(k+3)T$	5	$8T \log_5 n$
Pipelined Comparator	$(k+7)T/2$	7	$7T \log_7 n$

From the table we see that all the comparators using multi-valued architecture require more time than the binary architecture. So Binary search by binary architecture gives the best performance.

Chapter 6

Quick-sort with k partitions

This chapter is the most important chapter in the thesis. In this chapter quick-sort algorithm using k partitions has been presented in details. Different types of cost related with this algorithm have been calculated in the following section. The cost requirements in terms of time have been calculated for different types of proposed hardware for comparator. This time requirement has been calculated on some favourable assumptions. The optimum degree of multi-valued comparators for different hardware have been evaluated by differentiating expressions. A comparative study of different architectures for quick-sort algorithm has been tabulated at the end of the chapter.

6.1 Algorithm of Quick-sort with k partitions

The concept of dividing the problem space into k partitions in each recursive call can easily modify quick-sort algorithm. Although the algorithm of quick-sort recursive procedure will be slightly changed the partitioning algorithm will undergo a major change. Firstly we are going to write the quick-sort algorithm.

Procedure Quicksort (A, i, j, k)

A is an array. The elements from i to j will be sorted. k is the degree of partition.

$Partpoint[]$ – an array contains the beginning of each partition points.

Begin

if $(j - i + 1) \leq k$ then

Sort the elements by any well known method.

else

Sort the elements $A(i)$ to $A(i + k - 1)$ using any method

$m = Partition(A, i, j, k)$

for $m = i$ to $(k + 1)$ do

Quicksort($A, partpoint(m), partpoint(m + 1) - 1, k$)

endif

endProc

While partitioning some elements on an array, a different approach may be applied. For each element of the array an extra space is required. This extra space is only to store the result of the comparison, a value from 0 to $(k - 1)$. So only one byte is sufficient.

Now the array to be partitioned can be simulated as follows:

0	1		k-2			
						

$A(i)$
 $A(i+k-2)A(i+k-1)$
 $A(j)$

The array R contains the results of comparison. We can express this as follows:

$$\begin{aligned}
 R(k) &= 0, && \text{if } A(k) \leq A(i) \\
 &= 1, && \text{if } A(k) \leq A(i+1) \\
 &= \\
 &\vdots \\
 &= k-2, && \text{if } A(k) \leq A(k-2) \\
 &= k-1, && \text{if } A(k) > A(k-2)
 \end{aligned}$$

At first all the values of array R will be determined by k -valued comparison. This is the only cost required for comparing the values. The next step is partitioning the elements. For k partitions two more arrays of size k are required.

Consider the array S which contains the size of each partition.

$S(i)$ contains the number of elements in i th partition. This can be easily determined at the time of comparison. From these values of $S(i)$ an array $I(i)$ it can be easily determined by the following formula

$$\begin{aligned}
 I(0) &\leftarrow i \\
 I(i).\text{start} &\leftarrow I(i-1).\text{start} + S(i) \\
 I(i).\text{end} &\leftarrow I(i+1).\text{start} - 1 \\
 I(k-1).\text{end} &\leftarrow j
 \end{aligned}$$

Now consider a dummy array for swapping the elements. The array must be of size k , Let P be the name of the array.

$$P(i).\text{value} = \text{the element to be swapped}$$

$P(i).index =$ Destination partition of the element.

This queue is initialized by the misplaced elements in the array. Then these misplaced elements in the array, are placed in the original array by swapping another misplaced element in other partition.

6.2 Partitioning algorithm

Procedure partition(A, i, j, k)

Let $A[]$ be the array to be sorted. i, j be the starting and ending index of the elements to be partitioned. k is the degree of partition. $partpoint[]$ be the global array holding the starting and ending index of partition that will be used in quick-sort routine.

Begin

for $m \leftarrow i$ to $(i + k - 2)$ do

$R(i) \leftarrow (i - m)$ /* Results of initially sorted partitioned values */

$S(i - m) \leftarrow 1$ /* Initializing the sizes of partitions */

endfor

for $m \leftarrow i + k - 1$ to j do

$R(m) \leftarrow \text{compare}(A(m), A(i), A(i + 1), \dots, A(i + k - 2))$

$S(R(m)) \leftarrow S(R(m)) + 1$ /* counting the size of partition */

endfor

$I(0).start \leftarrow i$:

$partprint(0) \leftarrow i$

for $m \leftarrow 1$ to $(k - 1)$ do

$I(m).start \leftarrow I(m-1).start + S(i)$

$partpoint(m) \leftarrow I(m).start$

endfor

$I(k - 1).end \leftarrow j$

for $m \leftarrow (k - 1)$ to 1 do by -1

$I(m-1).end \leftarrow I(m).start - 1$

endfor

for $m \leftarrow 0$ to $(k - 1)$ do

$start \leftarrow I(m).start$

```

end ← I(m).end
while R(start) = m and start ≤ end do
    start ← start+1
endwhile
I(m).start ← start
if start ≤ end then
    P(m).value ← A(start)
    P(m).index ← R(start)
endif
endfor
need_for_further_looping ← 0
do
    for m ← 0 to (k - 1) do
        destn ← P(m).index
        if (destn ≥ 0) then
            start ← I(destn).start
            end ← I(destn).end
            A(start) ← P(m).value
            R(start) ← P(m).index
            while R(start) = destn and start ≤ end do
                start ← start + 1
            endwhile
            I(destn).start ← start
            if (start ≤ end) then
                P(m).value ← A(start)
                P(m).index ← R(start)
                need_for_further_looping ← 1
            else
                P(m).index ← -1
            endif
        endif
    endif
endif

```

end for

while (need_for_further_looping)

End procedure

6.3 Complexity of Multi-valued Quick-sort Algorithm

Consider that in each step k partitions are being done in every recursive call of quick-sort. For simplicity we can assume that the total number of elements in the array is k^l and in each step the array or sort space will be divided into k equal parts. So the sorting will be terminated at the level where all the sorting spaces are of size k . Now total multi-valued comparison cost can be expressed by the following recursive formula:

$$\begin{aligned}
 C_1(n, k) &= kC_1\left(\frac{n}{k}, k\right) + (n - k + 1) \\
 &= k\left\{kC_1\left(\frac{n}{k^2}, k\right) + \left(\frac{n}{k} - k + 1\right)\right\} + (n - k + 1) \\
 &= k^2C_1\left(\frac{n}{k^2}, k\right) + k\left(\frac{n}{k} - k + 1\right) + (n - k + 1) \\
 &= k^{l-1}C_1\left(\frac{n}{k^{l-1}}, k\right) + k^{l-2}\left(\frac{n}{k^{l-2}} - k + 1\right) + k^{l-3}\left(\frac{n}{k^{l-3}} - k + 1\right) + \dots + k^0(n - k + 1) \\
 &= k^{l-1}C_1(k, k) + n - (k - 1)k^{l-2} + n - (k - 1)k^{l-3} + \dots + n - (k - 1)k^0 \\
 &= (l - 1)n - (k - 1)[1 + k + \dots + k^{l-2}] \\
 &= (l - 1)n - (k - 1)\frac{k^{l-1} - 1}{k - 1} \\
 &= nl - n - \frac{n}{k} + 1 \\
 &= n \log_k n - n - \frac{n}{k} + 1
 \end{aligned}$$

To sort $(k - 1)$ elements by selection sort algorithm it requires $(k - 2)(k - 1)/2$ binary comparison. Total binary comparison cost can be expressed as follows:

$$\begin{aligned}
 C_2(n, k) &= kC_2\left(\frac{n}{k}, k\right) + \frac{(k - 2)(k - 1)}{2} \\
 &= k^2C_2\left(\frac{n}{k^2}, k\right) + k \times \frac{(k - 2)(k - 1)}{2} + \frac{(k - 2)(k - 1)}{2} \\
 &= k^{l-1}C_2\left(\frac{n}{k^{l-1}}, k\right) + \frac{(k - 2)(k - 1)}{2}(k^{l-2} + k^{l-3} + \dots + 1)
 \end{aligned}$$

$$\begin{aligned}
&= \frac{n}{k} C_2(k, k) + \frac{(k-2)(k-1)}{2} \times \frac{k^{l-1} - 1}{k-1} \\
&= \frac{n}{k} \frac{k(k-1)}{2} + \frac{(k-2)}{2} \left(\frac{n}{k} - 1 \right) \\
&= \frac{n(k-1)}{2} + \left(\frac{k}{2} - 1 \right) \left(\frac{n}{k} - 1 \right) \\
&= \frac{nk}{2} - \frac{n}{k} - \frac{k}{2} + 1
\end{aligned}$$

This is $O(n)$ complexity.

Total binary index comparison (byte comparison) can be expressed by the following recursive expression.

$$\begin{aligned}
&C_3(n, k) \\
&= kC_3\left(\frac{n}{k}, k\right) + n \\
&= k \left\{ kC_3\left(\frac{n}{k^2}, k\right) + \frac{n}{k} \right\} + n \\
&= k^{l-1} C_3\left(\frac{n}{k^{l-1}}, k\right) + n + n + \dots + n \quad [(l-1) \text{ s}] \\
&= (l-1)n \\
&= n(\log_k n - 1)
\end{aligned}$$

The cost of swapping depends on the statistical properties of observations. Swapping may be done in different steps. These are

- (i) Swapping for sorting the partitioning elements.
- (ii) Swapping for partitioning of sorting space.

To sort k elements by selection sort on the average it takes $k/2$ exchange or swap operations. So, total exchange operations is defined by

$$\begin{aligned}
&S_1(n, k) \\
&= kS_1\left(\frac{n}{k}, k\right) + \frac{k-1}{2} \\
&= k \left[kS_1\left(\frac{n}{k^2}, k\right) + \frac{k-1}{2} \right] + \frac{k-1}{2}
\end{aligned}$$

$$\begin{aligned}
&= k^{l-1} S_1(k, k) + \frac{k-1}{2} (k^{l-2} + k^{l-3} + \dots + 1) \\
&= \frac{k^l}{2} + \frac{k-1}{2} \times \frac{k^{l-1} - 1}{k-1} \\
&= \frac{1}{2} \left(n + \frac{n}{k} - 1 \right)
\end{aligned}$$

Total data transfer operation required is defined by

$$T_1(k, n) = \frac{3}{2} \left(n + \frac{n}{k} - 1 \right)$$

To partition sorting space it requires a very large number of swapping operations. Let us assume that we are going to make k partitions. So probability of an element belongs to its own partition is $1/k$. So, for partitioning n elements total number of swapping required is $\left(\frac{k-1}{k}\right)n$.

The data transfer requirement for partitioning is defined by the following expression

$$\begin{aligned}
&T_2(n, k) \\
&= kT_2\left(\frac{n}{k}, k\right) + \frac{2(k-1)n}{k} \\
&= k \left\{ kT_2\left(\frac{n}{k^2}, k\right) + \frac{2(k-1)n}{k} \right\} + \frac{2(k-1)n}{k} \\
&= k^{l-1} T_2\left(\frac{n}{k^{l-1}}, k\right) + \frac{2(k-1)n}{k} + \frac{2(k-1)n}{k} + \dots + \frac{2(k-1)n}{k} \\
&= 2(l-1) \frac{k-1}{k} n \\
&= 2(\log_k n - 1) \frac{k-1}{k} n
\end{aligned}$$

In every call of quick-sort $k-1$ data will be loaded to the registers. So total data loading requires for k -nary comparison operation is defined by the following expression.

$$\begin{aligned}
&L(n, k) \\
&= kL\left(\frac{n}{k}, k\right) + (k-1) \\
&= k^2 L\left(\frac{n}{k^2}, k\right) + k(k-1) + (k-1) \\
&= (k-1) \{1 + k^2 + \dots + k^{l-2}\} \\
&= (k-1) \times \frac{k^{l-1} - 1}{k-1} = \frac{n}{k} - 1
\end{aligned}$$

6.4 Time Requirement in Quick-sort with k -partitions

There are several types of cost related with quick-sort algorithm with k -partitions. We have to calculate and combine all of them. The related costs are discussed below.

- (i) Data loading to registers for k -nary search. If loading is done by pipelining processor $(k - 1)$ data in register requires $2T + (k - 2)T = kT$ unit time.
- (ii) Each binary comparison requires $7T/2$ unit time as shown in section 4.8.1.
- (iii) k -nary comparison. It requires different time for different types of architecture. For example by using bit comparison comparator it requires T units time for each compare instruction. Parallel comparator requires $2T$ units time for each compare instruction. In pipelining architecture there is no provision to pre-load the operands. So it requires no data loading time but $(k+7)T/2$ units time for k -nary comparison. Binary comparator requires $(k+1)/2 - 1/k$ binary comparison for k -nary comparison.
- (iv) Data transfer cost. It requires $2T$ unit time.
- (v) Byte comparison. This type of comparison can be ignored as it requires very small time comparing floating point numbers.

On the basis of the above mentioned parameter the time requirement for Quick-sort by different architecture is discussed in the next subsections.

6.4.1 Time requirement of Quick-sort with k -partitions using Binary comparator

Total time requirement can be defined by the following expression.

$$\begin{aligned}
 C_{\text{binary}}(n, k) &= C_1(n, k) \times \left\{ \frac{k+1}{2} - \frac{1}{k} \right\} \times \frac{7T}{2} + C_2(n, k) \times \frac{7T}{2} + T_1(n, k) \times 2T + T_2(n, k) \times 2T \\
 &= \left(n \log_k n - n - \frac{n}{k} + 1 \right) \times \left\{ \frac{k+1}{2} - \frac{1}{k} \right\} \times \frac{7T}{2} + \left(\frac{nk}{2} - \frac{n}{k} - \frac{k}{2} + 1 \right) \times \frac{7T}{2} \\
 &\quad + \frac{3}{2} \left(n + \frac{n}{k} - 1 \right) \times 2T + 2(\log_k n - 1) \left(\frac{k-1}{k} \right) n \times 2T
 \end{aligned}$$

Now differentiating both the side with respect to k the following equation can be found after simplification.

$$\begin{aligned}
\frac{dC_{binary}}{dk} &= 0 \\
\Rightarrow \frac{7}{2} \left\{ -\frac{n \log_c n}{k(\log_c k)^2} + \frac{n}{k^2} \right\} \left(\frac{k+1}{2} - \frac{k}{2} \right) &+ \frac{7}{2} \left(\frac{n \log_c n}{\log_c k} - n - \frac{n}{k} + 1 \right) \left(\frac{1}{2} + \frac{1}{k^2} \right) \\
+ \frac{7}{2} \left(\frac{n}{2} + \frac{n}{k^2} - \frac{1}{2} \right) - \frac{3n}{k^2} - \frac{4 \log_c n}{k(\log_c k)^2} \left(1 - \frac{1}{k} \right) n &+ 4 \left(\frac{\log_c n}{\log_c k} - 1 \right) \frac{1}{k^2} n = 0
\end{aligned}$$

By solving this equation by bisection method for different values of n , we find optimal value of k for different values of n .

6.4.2 Time requirement of Quick-sort with k -partitions using bit comparison

Total time requirement can be defined by the following expression.

$$\begin{aligned}
C_{bin}(n, k) &= C_1(n, k) \times T + C_2(n, k) \times \frac{7T}{2} + T_1(n, k) \times 2T + T_2(n, k) \times 2T + L(n, k) \times \frac{kT}{k-1} \\
&= \left(n \log_k n - n - \frac{n}{k} + 1 \right) \times T + \left(\frac{nk}{2} - \frac{n}{k} - \frac{k}{2} + 1 \right) \times \frac{7T}{2} \\
&+ \frac{3}{2} \left(n + \frac{n}{k} - 1 \right) \times 2T + 2(\log_k n - 1) \left(\frac{k-1}{k} \right) n \times 2T + \frac{k-1}{k-1} kT
\end{aligned}$$

Now differentiating both the sides with respect to k the following equation can be found after simplification.

$$\begin{aligned}
\frac{dC_{bin}}{dk} &= 0 \\
\Rightarrow \left\{ -\frac{n \log_c n}{k(\log_c k)^2} + \frac{n}{k^2} \right\} &+ \frac{7}{2} \left(\frac{n}{2} + \frac{n}{k^2} - \frac{1}{2} \right) - \frac{3n}{k^2} - \frac{4 \log_c n}{k(\log_c k)^2} \left(1 - \frac{1}{k} \right) n \\
+ 4 \left(\frac{\log_c n}{\log_c k} - 1 \right) \frac{1}{k^2} n &+ \frac{1-n}{(k-1)^2} = 0
\end{aligned}$$

By solving this equation by bisection method for different values of n we find optimal value of k for different values of n .

6.4.3 Time requirement of Quick-sort with k -partitions using parallel comparator

Total time requirement can be defined by the following expression.

$$\begin{aligned}
 C_{par}(n, k) &= C_1(n, k) \times 2T + C_2(n, k) \times \frac{7T}{2} + T_1(n, k) \times 2T + T_2(n, k) \times 2T + L(n, k) \times \frac{kT}{k-1} \\
 &= \left(n \log_k n - n - \frac{n}{k} + 1 \right) \times 2T + \left(\frac{nk}{2} - \frac{n}{k} - \frac{k}{2} + 1 \right) \times \frac{7T}{2} \\
 &\quad + \frac{3}{2} \left(n + \frac{n}{k} - 1 \right) \times 2T + 2(\log_k n - 1) \left(\frac{k-1}{k} \right) n \times 2T + \frac{k-1}{k-1} kT
 \end{aligned}$$

Now differentiating both the sides with respect to k the following equation can be found after simplification.

$$\begin{aligned}
 \frac{dC_{par}}{dk} &= 0 \\
 \Rightarrow 2 \times \left\{ -\frac{n \log_e n}{k(\log_e k)^2} + \frac{n}{k^2} \right\} + \frac{7}{2} \left(\frac{n}{2} + \frac{n}{k^2} - \frac{1}{2} \right) - \frac{3n}{k^2} - \frac{4 \log_e n}{k(\log_e k)^2} \left(1 - \frac{1}{k} \right) n \\
 &\quad + 4 \left(\frac{\log_e n}{\log_e k} - 1 \right) \frac{1}{k^2} n + \frac{1-n}{(k-1)^2} = 0
 \end{aligned}$$

Solving this equation by bisection method for different values of n we find optimal value of k for different values of n .

6.4.4 Time requirement of Quick-sort with k -partitions using pipelining comparator

Total time requirement can be defined by the following expression.

$$\begin{aligned}
 C_{pipeline}(n, k) &= C_1(n, k) \times \left(\frac{k+7}{2} \right) T + C_2(n, k) \times \frac{7T}{2} + T_1(n, k) \times 2T + T_2(n, k) \times 2T \\
 &= \left(n \log_k n - n - \frac{n}{k} + 1 \right) \times \left\{ \frac{k+7}{2} \right\} T + \left(\frac{nk}{2} - \frac{n}{k} - \frac{k}{2} + 1 \right) \times \frac{7T}{2} \\
 &\quad + \frac{3}{2} \left(n + \frac{n}{k} - 1 \right) \times 2T + 2(\log_k n - 1) \left(\frac{k-1}{k} \right) n \times 2T
 \end{aligned}$$

Now differentiating both the sides with respect to k the following equation can be found after simplification.

$$\frac{dC_{pipeline}}{dk} = 0$$

$$\Rightarrow \left\{ -\frac{n \log_c n}{k(\log_c k)^2} + \frac{n}{k^2} \right\} \left(\frac{k+7}{2} \right) + \frac{1}{2} \left(\frac{n \log_c n}{\log_c k} - n - \frac{n}{k} + 1 \right)$$

$$+ \frac{7}{2} \left(\frac{n}{2} + \frac{n}{k^2} - \frac{1}{2} \right) - \frac{3n}{k^2} - \frac{4 \log_c n}{k(\log_c k)^2} \left(1 - \frac{1}{k} \right) n + 4 \left(\frac{\log_c n}{\log_c k} - 1 \right) \frac{1}{k^2} n = 0$$

By solving this equation by bisection method for different values of n we find optimal value of k for different values of n .

6.5 Optimal degree for Different Comparator Architectures in Quick-sort Algorithm

The following table shows the optimal value of k for different values of n from 100 to 100000000000. This has been found by applying bisection method of finding roots of equation.

Table 11 Optimal values of k for different multi-valued comparators.

n	Binary Architecture		Bit Comparison		Parallel comparator		Pipelining comparator	
	$k_{optimum}$	k_{suit}	$k_{optimum}$	k_{suit}	$k_{optimum}$	k_{suit}	$k_{optimum}$	k_{suit}
100	3.55	4	4.12	4	4.66	5	5.20	4
1000	3.38	3	4.95	5	5.65	6	5.88	6
10000	3.29	3	5.73	6	6.57	7	6.38	6
100000	3.23	3	6.45	6	7.40	7	6.77	7
1000000	3.20	3	7.13	7	8.20	8	7.07	7
10000000	3.16	3	7.79	8	8.95	9	7.32	7
100000000	3.15	3	8.41	8	9.66	10	7.51	8
1000000000	3.13	3	9.01	9	10.35	10	7.68	8
10000000000	3.12	3	9.59	10	11.02	11	7.82	8
100000000000	3.10	3	10.16	10	11.68	12	7.95	8

$k_{optimum}$ = Optimum value of k for a particular comparator architecture and problem space size.

$k_{suitable}$ = The physically suitable value k for which cost function is minimum and near $t_{optimum}$.

Table 12 shows the time requirement of different methods for different values of n at optimal strategy with respect to binary architecture. The terms used in the table is defined below.

t_b = Time requirement for quicksort with two partitions (traditional) using binary comparator.

t_{bin} = Time requirement for quicksort with k partitions using binary comparator.

t_{bit} = Time requirement for quicksort with k partitions using bit comparison.

t_{par} = Time requirement for quicksort with k partitions using parallel comparator.

t_{pipe} = Time requirement for quicksort with k partitions using pipelining comparator.

Table 12 Ratio of time requirement for multi-valued architecture with respect to binary quick-sort with binary architecture.

n	t_{bin}/t_b	t_{bit}/t_b	t_{par}/t_b	t_{pipe}/t_b
100	1.000	0.578	0.636	0.830
1000	1.025	0.512	0.572	0.811
10000	1.037	0.472	0.532	0.798
100000	1.044	0.445	0.503	0.788
1000000	1.049	0.424	0.480	0.781
10000000	1.052	0.407	0.463	0.775
100000000	1.055	0.394	0.448	0.771
1000000000	1.057	0.382	0.436	0.767
10000000000	1.058	0.372	0.425	0.763
100000000000	1.060	0.364	0.416	0.761

The table shows the performance of quick-sort algorithms theoretically. Our analysis shows that quick-sort with k partitions at optimal strategy is better than traditional quick-sort algorithm for any multi-valued comparator architecture. It also shows that the performance of comparator by bit comparison is the best and that of pipelining comparator is the worst. The reason bad performance of pipelining

comparator is its basic principle. It does not provide the facility of loading the sorted array in the register and so for each comparison reloading has to be done. The performance of parallel comparator is worse than comparator by bit comparison because parallel comparator requires more time for comparison. The performance of binary architecture for quick-sort with 3 partition is always worse than traditional algorithm. Better performance can be found by modifying the partitioning algorithm as mentioned in KAYKOBAD [9]. In this paper a special partitioning algorithm has been applied for only three partition and so it gives better performance even using binary comparators. For variable partitioning such algorithm is difficult to establish and a subject of future research.

Chapter 7

Experimental Results

In this chapter some experimental results have been tabulated after writing programs of quick-sort and k -nary search algorithm. The time requirements of these two algorithms shown in this chapter are all estimation. The environment of the experiment, methods of the experiment and the theoretical assumptions of the estimation have been mentioned in this chapter. The response of practical result with respect to theoretical result has been discussed at the end of the chapter.

7.1 *Environment of the Experiment*

Performance of the new algorithm and new architecture can be determined by writing programs. We have done experiments with a very large number of data. So 16 bit compilers are not sufficient to do such experiments. Any 32 bit compiler can serve the purpose correctly. Again the O/S which does not support virtual memory or huge memory is not capable to run program which uses large memory. Even careful programming is not enough for that purpose. For the above mentioned reasons we have run our program under Windows95 O/S platform. This O/S uses all the facilities of virtual memory model. The machine used for this purpose has the following specification

- Pentium 133MHZ Intel Processor.
- 3.1 GB HDD.
- 256KB cache.
- 32MB RAM.

This specification allows the Windows 95 to run smoothly.

The compiler used for programming is Microsoft Visual C++ 5.0. This is a 32 bit compiler and very flexible to write program and implement algorithms. This compiler has an integrated Developer Studio that allows the program to edit, write, debug and run programs. Microsoft Visual C++ 5.0 is a product of Microsoft corporation and runs under the Windows 95 platform. So we can easily write and run program under same environment.

7.2 *How was the experiment done?*

To perform the quick-sort operation an array has been initialized with random numbers. Then the algorithm of quick-sort described in section 6.1 has been implemented. This is actually the algorithm which use quick-sort with k partitions using binary comparators. We can easily find the time required by this algorithm. But this is not sufficient for our experiment. We have to compare the performance of different H/Ws that have been proposed in chapter 4 for quick-sort algorithm. But these H/Ws are not

available in PC. So, we have made an estimation of time requirement for four kinds of comparator architectures.

The data used here is double precision floating point numbers. At first writing a program which compares and transfers a large number of data we have calculated the time requirement for a binary comparison of current PC and data transfer time from one location of one array to another location of another array. The results we have found are given below.

Time requirement for one double precision floating point comparison = 0.000000609 sec.

Time requirement for one double precision floating point data transfer = 0.000000423 sec.

Now we can easily determine the time requirement of the following components.

Time requirement for k -nary comparison by bit comparison = $\frac{T}{7T/2} \times$ time required for one binary comparison.

Time requirement for k -nary comparison by parallel comparator = $\frac{2T}{7T/2} \times$ time required for one binary comparison.

Time requirement for k -nary comparison by pipelining comparator = $\frac{(k+7)T}{2} \times \frac{2}{7T} \times$ time required for one binary comparison.

Time requirement for data loading = $\frac{2T}{7T/2} \times$ time required for one binary comparison.

These parameters have been used to estimate the time requirement by different comparators during quick-sort.

We have determined the optimal value of k theoretically from the equations derived in previous chapters for different values of n . Then the quick-sort algorithm is allowed to run for different values of k . Each execution for different optimal value of k gives the following parameters.

- Total number of binary comparisons.
- Total number of k -nary comparisons.
- Total number of k -nary data load operations.
- Total number of data transfer operations.

As estimated time requirement for all these operations are known we can easily determine the estimated time cost for different architecture in quick-sort algorithm. This estimated value is more than enough to compare the performance of different architectures.

To compare the results of binary search almost similar process has been performed. The sorted array from the quick-sort algorithm has been used as the search space. Then a particular amount of random numbers have been generated and stored in an array. k -nary searching has been done for different values of $k_{optimal}$ with the data to be searched. We can get the following parameters from the execution of the k -nary search.

- Total number of binary comparison.
- Total number of k -nary search.

These two data is sufficient to estimate the time requirement of k -nary search by different multi-valued architectures.

7.3 Experimental Results

The following table shows the estimated time requirement to sort 100,1000.....,1000000 data using quick-sort with k partitions for different multi-valued comparators. The data used here have been initialized by random numbers of Microsoft Visual C++. The time requirement is actually average time requirement of 10 different sets of data of same size.

Table 13 Time requirement of quick-sort algorithm for different multi-valued architecture.

Number of observations. n	Binary quick sort t_b	Binary architecture t_{binary}	k -valued architecture using bit comparison t_{bit}	k -valued parallel comparator. $t_{parallel}$	k -valued pipelined comparator $t_{pipelinc}$
100	0.000559	0.000861	0.000466	0.000512	0.000669
1000	0.009144	0.015388	0.006137	0.006855	0.009849
10000	0.120208	0.193927	0.073959	0.083109	0.125501
100000	1.504866	2.453354	0.819259	0.930527	1.471242
1000000	17.435128	44.755805	9.147047	10.500548	18.682495

From the table shown above it has been noticed that for smaller value n the time requirement becomes 20 times higher than that of previous value of n . This ratio becomes lower as the value of n increases. The complexity of quick-sort is $O(n \log_k n)$. So if the value of n becomes 10 times the complexity should be $10 \log_{10} 10n$ times higher. This value is near 20 for $n=10$ and as n increases it becomes near 10. The result also shows the achievement of the performance of quick-sort algorithm with the introduction of multi-valued architecture. Among the three proposed architectures the comparator with bit comparison shows the best performance. Besides this, it has lower optimal degree of multi-valued architecture than other two architectures. Actually it offers higher performance with lower cost. So our experimental result complies with the theoretical achievement here. The practical performance of pipelined processor does not comply with our theoretical result. This is only because of our assumption. The time requirement for data transfer was assumed $2T$ roughly but in practical case it seems to be higher than this. So practically pipelined processor shows less effective performance than that of traditional quick-sort with 2 partitions.

From Theoretical result we have found that the cost of quick-sort by bit comparison is nearly $1/3$ of traditional quick-sort using binary architecture. But the practical result shows that the ration is $1/2$. There may be several reasons for this variation between theoretical and practical result. These are listed below.

- (i) The analysis we have done is on the basis of most favorable case. This may not represent the practical or average case situation.
- (ii) The ratio is higher for smaller values of n . But for higher values of n it becomes lower. It does indicate that the set of observations approach to symmetrical distribution for higher values of n . We hope that for very large values of n it will fully comply with the theoretical outcome.
- (iii) In the experimental result we have found an estimated result. This is not a true result from a real program which uses multi-valued architecture. Even if somebody uses the multi-valued architecture the theoretical result will not be proved fully correct. We have ignored several cost in the analysis. These are cost of index calculation and cost of byte comparison for partitioning. These are the trivial costs and if we consider the double precision floating point data then these can be hopefully ignored.
- (iv) The additional space requirement for storing the result of multi-valued comparison is one of the important cost related with the newly proposed algorithm.

Table 14 Time requirement of k -nary search algorithm for different multi-valued architecture.

Size of search space and number of observations have been searched n	Binary search with Binary architecture t_b	k -nary search with Binary architecture t_{binary}	k -valued architecture using bit comparison t_{bit}	k -valued parallel comparator. $t_{parallel}$	k -valued pipelined comparator $t_{pipeline}$
100	0.000413	0.000524	0.000643	0.001414	0.001725
1000	0.006042	0.007669	0.009745	0.024129	0.029110
10000	0.078007	0.103417	0.125792	0.306922	0.366021
100000	0.813527	1.317981	1.677988	4.186050	4.935621
1000000	7.959615	15.275028	19.953241	48.875601	59.817610

Table 14 shows the estimated time requirement to search 100,1000.....,1000000 data using k -nary search for different multi-valued comparators in a sorted array of size 100,1000,.....,1000000 respectively. The data used here has been initialized by random numbers of Microsoft Visual C++. The time requirement is actually average time requirement of 10 different sets of data of same size.

In the experiment we have done n number of k -nary comparisons. So the complexity becomes $O(n \log_k n)$. So same response is expected as quick-sort for the ratio of time requirement. From the table shown above it has been noticed the response is far better than the quick-sort algorithm. This may be because of additional terms in the complexity function of quick-sort. The data shown in the table justifies the theoretical result that multi-valued search is not better after introducing multi-valued architecture. The reason behind it is that the multi-valued comparison is not used with its full efficiency and the data is loaded for every level of searching algorithm. Pipelined processor shows the worst performance in searching algorithm as well as sorting algorithm. Actually for binary comparison it gives worse performance than dedicated binary comparator without pipeline. So we can reach into such a conclusion that pipelining idea in comparator is not successful for both sorting and searching algorithm.

Chapter 8

Further Improvement

This chapter discusses the shortcomings and further development that may be possible future research plan on this topic. Only plans are describe here with few words. Eight research proposals have been presented here. Some of them may be feasible and some are not feasible till the invention of suitable electronic devices.

8.1 *Shortcomings of the thesis*

- In our research we have made some assumptions. These assumptions have been done by doing experiment in normal PC. The assumptions made may not be valid in other machines and computers.
- In the proposed H/W design the architecture we have proposed is only for positive integer comparison. This is actually a conception. Generally we are interested in comparison of large and complex data. For example, double precision data type. We have made an assumption that the operations on double precision numbers are comparable with normal integer data types.
- The estimation of time requirement in sorting algorithm has been done for most favorable cases. Now compilers use different types of optimization. Without knowing details of the compiler processing the exact estimation is not possible.
- While determining the complexity we have measured the complexity for favorable cases. This is not average case. The average case analysis is very difficult because we have to consider all possible combinations. Besides these, the complexity of index comparison has been ignored.
- Random number generator of C compiler has been used here. Random numbers for particular statistical model should be applied for worst case and best case performance analysis or analysis of a particular type of data.

8.2 *Future Research Plan*

8.2.1 *New Device to implement ternary logic:*

Transistor is a device which has two stable states. This is the basic element of modern digital electronic circuits. Such circuit is not available till now that gives 3 stable output states. This type of device is a hot topic of research for the electronic engineers. Such device can change the idea of information technology and at the same time the world also. If such device is invented then the internal circuit of logic gates i.e., AND, OR, NOT, NAND gates will be changed. The input, output, noise margin and operating conditions will be defined again with new values. These are the future research topics for multi-valued logic circuit.

8.2.2 *True Multi-valued Architecture*

The architecture we have designed and discussed in the thesis is pseudo multi-valued architecture. The true multi-valued architecture is that which gives multiple output for a particular decision. That is the comparator must have more than one output for multi-valued comparison. In the thesis such structure has not been described and the algorithm for that type of architecture has not been derived. The investigation of such architecture is a potential research plan.

8.2.3 *VLSI design of ternary memory cell*

Only a ternary device is not sufficient to design ternary system. We have to make cost benefit analysis of the proposed architecture or system. If we introduce ternary memory system, i.e. the bits will store 3 logic levels instead of 2 then the memory cell will be changed. It will require more space individually. Additional circuit may be needed to read, write and refresh the contents of the memory. So total space requirement will be changed and time requirement for different operations are not same. A detailed VLSI design of this type of memory architecture is compulsory to determine the power requirement, delay and space requirement of the circuit. Then it will be helpful to take decision on introducing ternary memory system.

8.2.4 Double Precision Floating Point Comparator

Double precision comparators are not easy like integer comparator. We have to compare the separate parts of the double precision number separately. For designing multi-valued double precision floating point comparator we have to be careful. Additional circuits must be added for such type of comparators.

8.2.5 Detailed circuit diagram

The first step of system design is block diagram. The second step is circuit diagram. The final step is VLSI layout design of different layers. The second steps i.e., detailed circuit diagram of multi-valued integer and double precision floating point comparators may be a research project. There are so many circuit simulators which is very helpful to justify the circuit after designing.

8.2.6 Cost Benefit Analysis

It is obvious that introducing multi-valued system requires additional cost and additional space. But we also found that the more the degree of multi-valued comparator the more benefit we can earn by using it for quick-sort algorithm. There are so many algorithms that may use the multi-valued comparators successfully. These algorithms are the future development of the topic. Introducing some positive weights on quick response and negative weights on H/W complexity we can establish a formula which may be helpful for decision making of introducing multi-valued comparators.

8.2.7 Exact Simulation of Multi-valued Comparator

As we are not getting readymade multi-valued comparator, simulation is the only way of research. To simulate different comparators we have to simulate the work flow of the comparators. The integers and floating point numbers will be stored in a character array as a stream of 0s and 1s and the program will work only with 0s and 1s. The operations will be done on these bit streams only during the simulation. This may be the exact simulation of the processor. The time requirement of this simulation may be helpful to compare the architecture. Besides this, VHDL representation can serve the purpose also.

8.2.8 Perfect Analysis of Multi-valued Quick-sort using multi-valued comparator

The average case analysis of multi-valued quick-sort is very necessary to determine the perfect complexity of multi-valued quick-sort algorithm. Without this complexity analysis the performance criteria of multi-valued architecture cannot be estimated perfectly. Statistical probability analysis is also necessary for this purpose. Researchers with strong mathematical and statistical background may carry out such research. The average case complexity analysis of other multi-valued partitioning algorithms are also prospective future research area.

References

- [1] ALLEN C. M. and Givone D. D., *A minimization technique for multiple-valued logic systems*, pp. 182-184 IEEE transaction on Computers, Feb 1968.
- [2] BUTLER JON T. and SASAO TSUTOMU, *Multiple-valued Combinational Circuits with Feedback*, Proceedings of the 24th International Symposium on Multiple-valued Logic, May 1994.
- [3] GOBEL F., *On an Optimality Property of Ternary Trees*, Information and Control, pp10 – 26 Academic Inc., 1979.
- [4] HAYES J. P., *Computer Architecture and Organization*, 2nd Edition, Mc. Grow Hill International Editions, Computer Science Series, USA, 1988.
- [5] HELLERMAN L., *A catalog of three-variable OR-invert and AND-invert logic circuits*, pp. 198-223 IEEE transaction on Computers, June 1971.
- [6] HOROWITZ ELLIS and SAHNI SARTAJ, *Fundamental of Computer Algorithms*, Computer Science Press., USA, 1993.
- [7] KAYKOBAD M., *3 is more promising algorithmic parameter than 2*, to appear in International Journal of Computers and Mathematics with Applications. 1998.
- [8] KNUTH D. E., *The Art of Computer programming: Fundamental Algorithm*, Volume 1, Addison Wesley Publishing Company Inc., 1973.
- [9] KNUTH D. E., *The Art of Computer programming: sorting and searching*, Volume 3, Addison Wesley Publishing Company Inc., 1973.
- [10] MEGIDDO NIMROD, *Is Binary Encoding appropriate for the Problem-Language relationship*, Theoretical Computer Science Vol.19 pp. 337 – 341, North – Holland Publishing Company, 1982.
- [11] PINTER SHLOMIT S., *Embedding Ternary trees in VLSI arrays*, Information Processing Letters, pp 187-191 North – Holland Publishing. 1987.
- [12] SCHAFFER RUSSEL, *The Analysis of Heap Sort*, Journal of Algorithms, pp 76-100, Academic Press Inc., 1993.
- [13] SEDGEWICK R., *Quick-sort*, Computer Science Department, Stanford technical report, May 1975.
- [14] WESTE and ESHRAGHIAN, *Principles of CMOS VLSI Design*, Second Edition, VLSI Systems Series, AT&T, USA, 1993.

