MS Thesis Report

# Bit Parallel Computation of SBNDM4 Algorithm Using Scalable Parallel Hardware

*Submitted by*
Muhammad Tahir Rana (NUST201463704MPNEC45314F)

*Under the guidance of*
Dr Naeem Abbas

Fall, 2014. Submitted in partial fulfillment of the requirements for the award of the degree of:

MASTER of SCIENCE
in
Communication Engineering

Department of Electronic and Power Engineering
PAKISTAN NAVY ENGINEERING COLLEGE, NUST, KARACHI
January 26, 2017

# Acknowledgments

First of all, praise is to Allah, on whom all of us depend for nourishment and supervision. Secondly, I would like to be sincerely grateful to my supervisor Dr. Naeem Abbas for his consistent support, perfect supervision, meticulous suggestions and persistent criticism throughout the research work and his limitless patience during the correction phase of dissertation. With his well-timed and efficient involvement, the thesis can successfully transform to its final shape. I would also like to thank Dr. Ali Hanzala and Dr. Bilal Khan, for their continuous support, assistance and guidance. Last but not the least, I wish to express my thanks to all of those who have one way or other helped me in making research a success.

# Abstract

Exact String Search algorithms have been a very active research area since last 5 decades. The emphasis of research have mainly been on the speed at which an algorithm processes a given text. Bit-parallelism and character skipping capability are two of the main attributes defining the speed of processing. An algorithm's performance, along with other factors, is also dependent on the implementation, the length of target pattern and the platform on which the algorithm is implemented. SBNDM4 is one of the modern Exact String Search algorithms that works on the principal of non-deterministic suffix automation. This algorithm performs good against short to medium pattern lengths. FPGAs are the devices inherently favorable for the process requiring parallel processing. In this thesis we have demonstrated the implementation of SBNDM4 on one of well know FPGA using HDL. We have proposed an scalable experiment setup that although runs SBNDM4 instances, but can also be used for the implementation of other such algorithms in same manner. We have compared the results yielded by this experiment with [1]. Experimental results show that our proposed setup outperforms the system proposed in [1], even while working on much slower clock frequency.

# Table of contents

# List of Figures

This page is intentionally left blank.

# Chapter 1

# Introduction

*In this chapter we will look at the concepts of Hardware based process accelerators. Along the way accelerators based on different types of devices will be introduced. Special emphasis will be on FPGA based accelerators. In later sections we will see what is text searching, their applications and what are modern trends in the field of text searching algorithms. In the last of this chapter we will introduce SBNDMq.*

In this thesis we have done our work on a technology that lies on the cross roads of two broader technological fields. One is Hardware Accelerators(or high performance computing) and other being String Searching. So, at first, we are discussing hardware accelerators in detail.

## 1.1 Hardware Accelerators

In the paradigm of Embedded systems and Computer systems, the word acceleration refers to a piece of software or hardware, that speeds up certain computational processes.The High Performance Computation or Data process acceleration is a branch of computational science that aid's in solving complex or compute-intensive problems[2], or, aggregating computational power in such a way that delivers more than the situation where only a single Desktop or workstation was used to solve a big or complex problem[3].

Traditionally and conventionally, computer clusters constitute most of the these acceleration facilities. The idea of clusters stems from the notion that although microprocessor are being manufactured for general use, but by combining the power of many such processors, an aggregated processing performance may be attained. But with specialized processors and embedded technology development, this scenario is changing rapidly.

Nowadays accelerators based on circuits for specialized applications are gaining more and more space in scientific computation. Some of the examples are:

- Graphics Processing Units
- Cryptographic accelerators
- Math co processors(Floating-point and arithmetic)
- Accelerators for neural networks processing

## 1.2 String Search Algorithms

String Search, as its name suggest is a process which involves searching a particular pattern in a larger given data or strings. For example, if we are given a text T of length $n$, then a string search algorithm will search all occurrence of pattern P, of length $m$ out of this given text.

String matching may be broadly categorized as *exact* and *approximate* matching techniques[4]. In this thesis as we are working on exact matching techniques, we will not discuss approximate techniques any further.

One of the earliest exact string matching algorithm is Aho-Corasick algorithm[5]. This algorithm builds a finite state tree based on finite set of key words(dictionary). On each subsequent input, it performs matching for all key words(patterns), based on failure pointer, decides whether two subsequent states were discontinued. It is a scalable solution. The computational complexity of Aho-Corasick reaches $O(m + k)$, where $m$ is pattern length and $k$ is the total number of strings found[6].

Here it may be noted that there is no any single algorithm that works well against all lengths of alphabet. Similarly , different algorithms works well against different length of patterns [7]. So it may be beneficial that prior to use any algorithm, application requirements must be thoroughly studied. Decision about pattern length, alphabet size and number of bits per character must be determined before any other decision.

In this thesis we will mainly talk about the searching algorithms which search the text serially. Some of the algorithms are discussed below. Brute-Force(BF) algorithm is the simplest example of sequential searching algorithms. This algorithm tries to match the given pattern with all the positions in the given text to see whether the given pattern is present there or not. The worst case timing for this algorithm is $O(nm)$, as there are $O(n)$ text positions and also there are $O(m)$ characters in patterns.

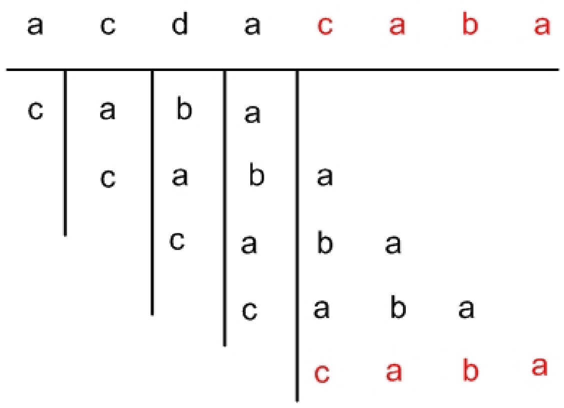


Figure 1.1: Brute-Force Algorithm example

In the figure 1.1, pattern of length four is being searched with the help of Brute-Force algorithm. The target pattern is *caba*. At every shift from left to right, the whole pattern is being matched with respective characters in text. This process continues until a total match takes place.

Knuth-Morris-Pratt (KMP)[8] algorithm is another one of the early works in the field of string search algorithms. This algorithm has a linear worst case performance. This algorithm essentially scan a given string, character by character, building an automaton, independent of the string it self. KMP constructs a prefix graph, which may be easily implemented using any standard finite state machine. The worst case performance of KMP is $O(n)$, where $n$ is length of text or string.

The basic working principal of this is taking any text character by character. When no match is found at first character, the window slides a single character towards right. When any single character mismatches window shifts itself till that character. This process yields an scanning speed not significantly faster than BF.

The figure 1.2 shows an example of KMP, where a pattern *abacab* is being searched. There are two cases depicted, when last character mismatches, and last character of window is either i) same as first character of pattern, ii)or not.

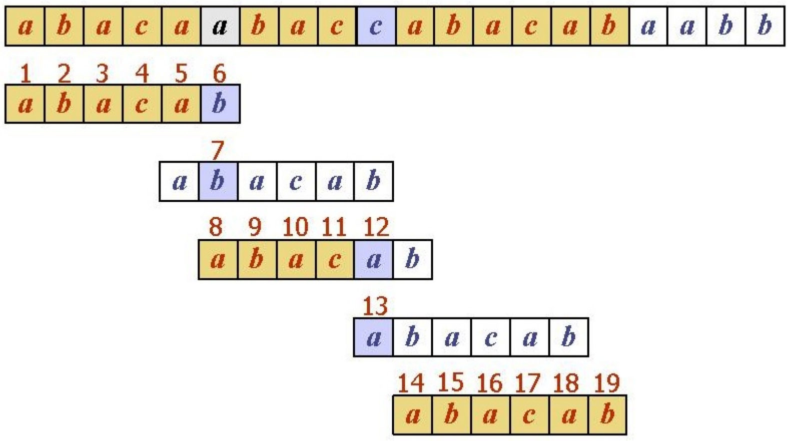


Figure 1.2: Knuth-Morris-Pratt Algorithm example

Boyer-Moore(BM)[9], is a relatively better algorithm which allows character skipping. It was also unique because it was the first algorithm that provided sub-linear timing solution[10].

This algorithm works based on the principal of *suffix* automaton, i.e. it compares the pattern from right to left. Character window matches its self against the given text. When all or some characters in a window are mismatched, moving from right to left, the first character causing mismatch is observed. If this character doesn't fall at any position of target pattern, then the window is shifted whole past the current text characters in window. But if the character has an occurrence at any position in the pattern than place the window with that character aligned in both the window and text. This algorithm perform on average as $O(n\log(m)/m)$, and the worst case performance is $O(mn)$.

Shift-Or [12] or Shift-And[13] algorithms were the first utilizing bit parallelism in string search. Bit parallelism is a technique taking advantage of the individual bits present in a word. If one uses this fact wisely, the number of operations performed may be reduced by a factor of word size. If *W* is the word size, then the worst case performance of Shift-Or algorithm is $O(mn/W)$. In this algorithm a mask table is generated for each character in pattern , say $B[T_j]$, first, where the number of bit in a mask are equal to the word size, and for a character occurring in pattern, the corresponding bit position is set. An state register, say D, is first initialized with all bit set. The consecutive calculations are done using following formula:

$$D' \leftarrow (D \ll 1)|B[T_j] \tag{1.1}$$

Above mentioned equation shows that every consecutive character is ORed and is saved in $D$, and hence $D'$. Once the most significant bit becomes reset, indicates the detection of a pattern.

## 1.3 Algorithmic Background

SBNDM4 is a variation of SBNDMq algorithm, which was first proposed by Branislav Durian, Jan Holub, Hannu Peltola and Jorma Tarhio in their paper [1]. SBNDM stands for *Simple Backward Non-deterministic Directed Acyclic Word Graph Matching*. 4 in SBNDM4 represents q, which we will see shortly. SBNDM4 is an exact string matching algorithm. This algorithm searches for a single pattern at a time. SBNDM4 performs best against short and medium length patterns.

Shift-Or algorithm was the work that paved the way for many other better and more efficient algorithms. First of the works that was an improvement of same Shift-Or algorithm was BNDM[14] algorithm by Navarro and Raffinot. This algorithm used Shift and AND operation. In the year 2003, TNDM(here T stands for Two-way)[15] was proposed by Peltola and Tarhio. In the same paper they also proposed Simplified-BNDM or SBNDM. For going any further, we will first see some background algorithmic concepts, then in section 1.6 we will see what are BNDM, SBNDM and SBNDMq, then we will proceed towards SBNDM4 concept.

## 1.4 DAWG and Suffix Automation

DAWG stands for Deterministic Acyclic Word Graph. The concept of DAWG[16] is very vital and forms the basis of many other text search algorithms.

Historically in almost all string search problems we are given a text T and we are asked whether a pattern P exists in that text or not. Conventionally we subject the given data to some sort of pre-processing that prepares our data for further searching process[5, 8, 9]. As searching and matching still takes time which is proportional to word length. This method becomes even more time consuming when different pattern are matched against a given text. It is highly desirable to pre-process data into a model or form, so that processing times becomes proportional to pattern length, not the word length, as we know that the length of pattern is always less than or equal to word length. The data formed or pre-processed in such a manner are called *Suffix Tree* structures [17, 18, 19, 20, 21, 22]. This type of data structure is widely applicable in a broad spectrum of application pertaining to string search, and for examples please see [23, 24, 25, 26, 27, 28]. By doing and following these guides, we build a structure that is partial at each branch. By partial we mean that every branch does not consider every possible character in pattern, and that's how we build a *Suffix Automaton*.

In Suffix tree we deal with states, and the $T_o$ being the initial state. As characters are passed to the automaton, states are changed and final states are called Terminal states, this is where we don't proceed for any further states. As we move along from Initial state to Terminal state, we get a Suffix of original pattern at every successive state. These are Directed graphs, because one can move only in predefined direction from one state to another state. Similarly, the word *acyclic* comes from the fact that the whole process is not cyclic, that mean it is possible moving from one state to another, but not in reverse order.

### Example of making a DAWG

In this example we will see how to build a DAWG. Suppose we want to build a DAWG which represents five words: DOG, DOT , TAB, TI and TAR. Right from the initial state if we receive first letter as D, then we will draw a branch towards state S1, showing a transition from S0 to S1. At state S1, only character O will be mapped for a transition to state S2, ignoring all other characters. If the next character is T, then our graph will reach S3, which is Terminal state. If from state S2, next character is G, then a transition from state S2 to S4 will take place.

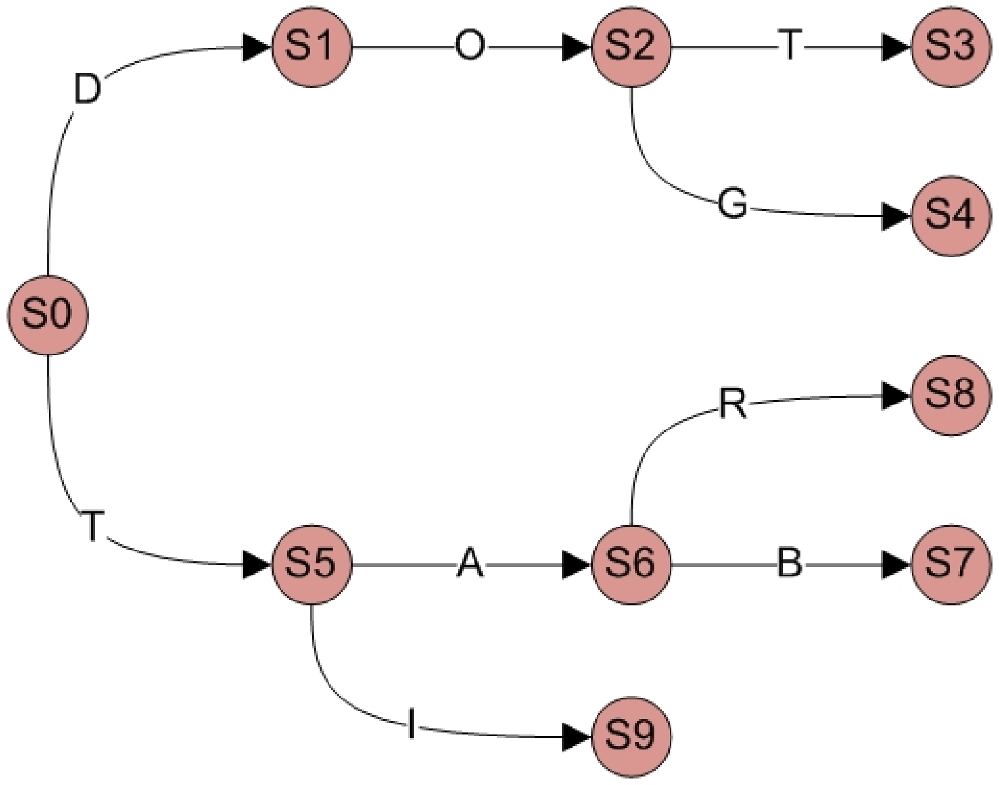


Figure 1.3: A DAWG example

S4 is a terminal state. Now again at initial state S0, if the input character is T, a transition from S0 to S5 will take place. If next character is A then a transition from S5 to S6 will take place. Or if at state S5 the next character input is I, then a transition from S5 to S9 will take place. S8 is a terminal state. At state S6 if next character input is R, a transition from S6 to S7 will take place. Similarly, at state S6, if next character input is B, a transition from S6 to S7 will take place. S8 and S7 are both terminal states. It may be noted that for a pattern TAB, the suffixes are TAB, AB and B.

## 1.5 Bit Parallelism

In the paper [12] a newer approach for a pattern or string detection was introduced. This approach was based on *Bit-parallelism* [29]. This technique took advantage of performing operations and state checking of different bits within the computer byte or word. This property , if used with skill can reduce overall operations performed by microprocessor by a factor of word length of the processor.

We will describe Bit-parallelism deployed in BNDM[30] which is the basis for SBNDM4. Bit parallelism takes the advantage of already existing parallelism within the bits of computer words. BNDM searches a given text by sliding a window over the text. The state of searching is updated in D, we call it the state register, each time a character is checked. D contains m number of bits which are equal to number of characters in pattern. Let suppose D = $d_m......d_1$.

Each time the window is placed just after *pos*, it searches the window in reverse direction, that is backwards. Searching is done by using DAWG, from $T_{pos+1}$ to $T_{pos+m}$. If m number of characters are checked, and D is non-zero, then its a pattern finding. But if D becomes zero before reaching m number of characters or at reaching m number of characters then further characters are not searched and a jump decision is taken. We initialize D with $1^m$, where m signifies number of 1s (e.g. $1^9 = 111111111$). It may be further noted that when ever $d_m$ is 1, we have a prefix of the target pattern. The prefix found minus the pattern length is the next window position(and hence the parameter *last*). Some parameters are depicted in figure 1.4 .

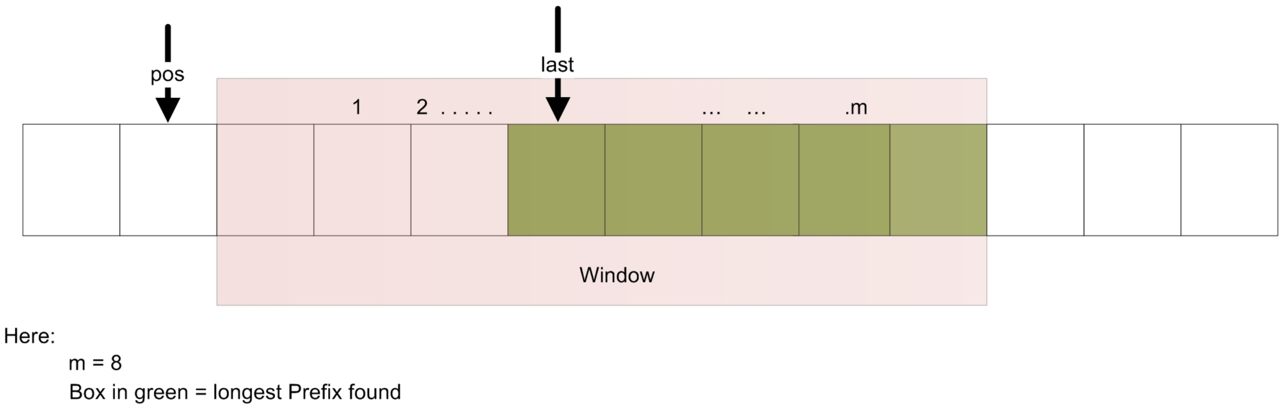


Figure 1.4: Bit Parallelism in BNDM and Window function

In figure above a window of size 8 is given. That asserts that m=8, and pattern length is 8. The *last* parameter is the windows position, where the longest prefix of a pattern was found. *pos* is the position just before where the window was last scanning. We will further explore it in section 1.6.

Some of the notable string search algorithms involving bit-parallelism are listed in table 1.1 .

| | | | |
|---|---|---|---|
| SO | Shift-Or | [32] | 1992 |
| SA | Shift-And | [32] | 1992 |
| BNDM | Backward-Nondeterministic-DAWG-Matching | [10] | 1998 |
| BNDM-L | BNDM for Long patterns | [14] | 2000 |
| SBNDM | Simplified BNDM | [15] | 2003 |
| TNDM | Two-Way Nondeterministic DAWG Matching | [15] | 2003 |
| LBNDM | Long patterns BNDM | [15] | 2003 |
| SVM | Shift Vector Matching | [15] | 2003 |
| BNDM2 | BNDM with loop-unrolling | [33] | 2005 |
| SBNDM2 | Simplified BNDM with loop-unrolling | [33] | 2005 |
| BNDMBMH | BNDM with Horspool Shift | [33] | 2005 |
| BMH-BNDM | Horspool with BNDM test | [33] | 2005 |
| FNDM | Forward Nondeterministic DAWG Matching | [33] | 2005 |
| BWW | Bit parallel Wide Window | [34] | 2005 |
| FAOSO | Fast Average Optimal Shift-Or | [35] | 2005 |
| AOSO | Average Optimal Shift-Or | [35] | 2005 |
| BLIM | Bit-Parallel Length Invariant Matcher | [36] | 2008 |
| FSBNDM | Forward SBNDM | [37] | 2009 |
| BNDMq | BNDM with q-grams | [1] | 2009 |
| SBNDMq | Simplified BNDM with q-grams | [1] | 2009 |
| UFNDMq | Shift-Or with q-grams | [1] | 2009 |
| SABP | Small Alphabet Bit-Parallel | [38] | 2009 |
| BP2WW | Bit-Parallel2 Wide-Window | [39] | 2010 |
| BPWW2 | Bit-Parallel Wide-Window$^2$ | [39] | 2010 |
| KBNDM | Factorized BNDM | [40] | 2010 |
| KSA | Factorized Shift-And | [40] | 2010 |

Table 1.1: Some of the notable bit-parallel algorithms [31].

## 1.6 SBNDMq

SBNDMq is an improvement in SBNDM, and was first propsed in [1]. SBNDMq proposes that q number of characters should be tested before deciding to proceed for further testing or jumping. This technique enables effort saving in address generation and in our case we have chosen q to be 4 and we call it q-gram.

### 1.6.1 Understanding SBNDMq

Before jumping into pure maths, lets first discuss some of the mathematical notation we will be using for describing this algorithm. We assume that a pattern P = $p_1p_2p_3p_4.....p_m$ is to be found out of a given text T = $t_1t_2t_3t_4.......t_n$. The alphabet that may exist within the text is given by the symbol $\Sigma$. We have to search all the text positions *i* such that $t_it_{i+1}....t_{i+m-1}$ = $p_1p_2.....p_m$. Symbol "|" represents bitwise OR operation. The symbols "&" represents bitwise AND operation. Symbols $\ll$ and $\gg$ represents left and right shifts respectively. The computer word width is denoted by symbol $w$.

**Algorithm 1** SBNDMq

```
1:  for $a \in \Sigma$ do $B[a] \leftarrow 0$ end for
2:  for $j \leftarrow 1..m$ do
3:      $B[p_j] \leftarrow B[p_j] \mid (1 \ll (m - j))$
4:  end for
    Compute $s_o$ with Alg. 2
5:  $i \leftarrow m - q + 1$
6:  while $i \leq n - q + 1$ do
7:      $D \leftarrow F(i, q)$
8:      if $D \neq 0$ then
9:          $j \leftarrow i - (m - q + 1)$
10:         do
11:             $i \leftarrow i - 1$
12:             $D \leftarrow (D \ll 1)\ \&\ B[t_i]$
13:         while $D \neq 0$
14:         if $j = i$ then
15:             $report\ occurrence\ at\ j + 1$
16:             $i \leftarrow i + s_o$
17:         end if
18:     end if
19:     $i \leftarrow i + m - q + 1$
20: end while
```

In algorithm 1, the basic mathematics in form of a pseudo code of the SBNDMq is given. In first line all alphabets are stored as pure zero or all zero words. This is because in next phase we will (shown in line three) we will generate bit masks for all the characters present in the pattern. In line five, a jump size is shown by the equation m - q + 1 equation. For our case this is 9 - 4 + 1 = 6. Whenever there is no prefix detection or after the detection of complete pattern, current first address of window is updated with this jump size. We will see the details in example discussed in section 1.6.2 . In line 7, the state register is updated in such a way so that:

$$F(i,q) = (B\ [t[i]] \ll q-1)\ \&\ (B[t[i-1]] \ll q-2)\ \&..\&\ (B[t[i-q+1]] \ll q-q) \quad (1.2)$$

If D $\neq 0$, which is checked in line 8, then the value of *i* minus the jump value is updated in j, which is also the expected start of pattern minus one, and the value of *i* is decremented. In line 12 the same process as in line 7 is done, except for this time only current character w.r.t *i* is considered. Again the condition D $\neq 0$ is checked. If still the value is non zero, then program enters the while loop, which decreases the value of *i* at every iteration. In line 14, the condition is presented where the value of *i*, the character index, touches j, expected start of pattern. This part will only be reached when part presented (while condition)in line 13 will be false. If at this point j = *i* then an occurrence is reported at position j + 1 and the value of *i* is updated with *i* + $s_o$. $s_o$ will be now the longest prefix found plus the remaining part of pattern. The process for calculating $s_o$ is given in algorithm 2.

**Algorithm 2** Calculating $s_o$ as a sub part of SBNDMq

```
1: $S \leftarrow B[p_m]; s_o \leftarrow m$
2: **for** $i = m - 1 \rightarrow 0$ **do**
3:     **if** $S\ \&\ (1 \ll (m-1)) \neq 0$ **then** $s_o \leftarrow i$ **end if**
4:     $S \leftarrow (S \ll 1)\ \&\ B[p_i]$
5: **end for**
```

Now back to line 9, if D is zero then we will jump directly to line 19, where it is just incremented with jump length and that is how this algorithm goes scanning a text.

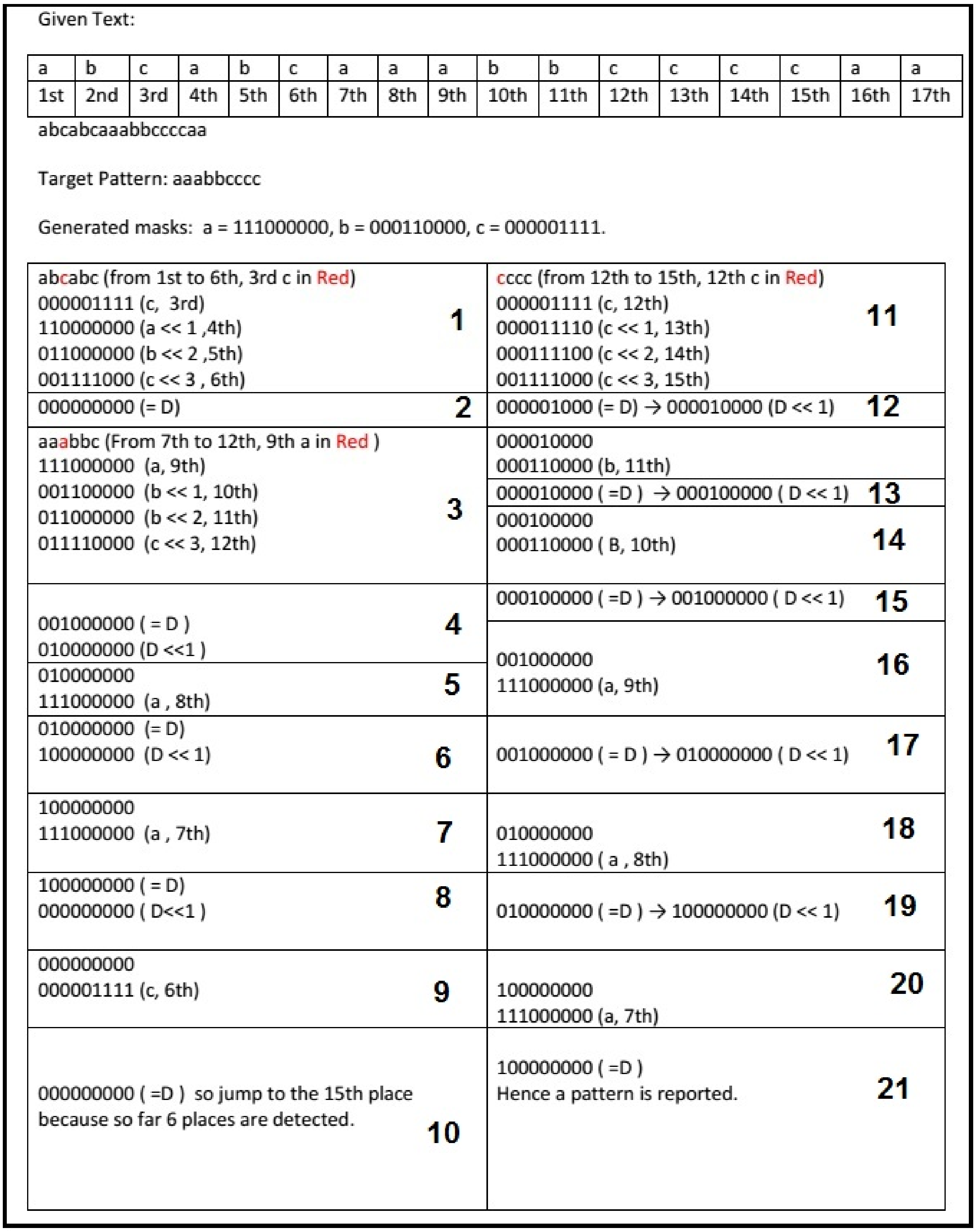
**Why backwards?**

The idea of scanning in forward direction seems simpler and straightforward. Specially when we are dealing with English text or any other data, character by character or word by word. But its an inherent property of most of real life applications data that patterns are more unique when observed in backward direction than when observed in forward direction. A real life example may be a list of Land Line Telephone numbers, when searched for a particular Telephone number. This property saves a considerable amount of effort, number of operations for identification of a target pattern and overall resources utilization. Utilizing this property researchers devised many new algorithms which rely on scanning data backwards such as BNDM.

### 1.6.2 How SBNDM4 works

In previous sections we saw how SBNDMq algorithm works. In this section we will see how that all applies to the operation of SBNDM4. Lets suppose we are given a text of some arbitrary length. We take a chunk of 17 characters to extract a pattern of length 9. Each character is represented by 9 bits.

In pre-processing phase where masks for individual characters are generated, only all zero pattern are generated for those characters which are sure not be the part of target pattern. Suppose we are targeting a pattern consisting of three alphabets a, b and c and the given text is abcabcaaabbccccaa and target pattern is aaabbcccc. In pre-processing phase we will generate



Given Text:

| a | b | c | a | b | c | a | a | a | b | b | c | c | c | c | a | a |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1st | 2nd | 3rd | 4th | 5th | 6th | 7th | 8th | 9th | 10th | 11th | 12th | 13th | 14th | 15th | 16th | 17th |

abcabcaaabbccccaa

Target Pattern: aaabbcccc

Generated masks: a = 111000000, b = 000110000, c = 000001111.

| | |
|---|---|
| abcabc (from 1st to 6th, 3rd c in Red)<br>000001111 (c, 3rd)<br>110000000 (a << 1 ,4th)<br>011000000 (b << 2 ,5th)<br>001111000 (c << 3 , 6th) **1** | cccc (from 12th to 15th, 12th c in Red)<br>000001111 (c, 12th)<br>000011110 (c << 1, 13th)<br>000111100 (c << 2, 14th)<br>001111000 (c << 3, 15th) **11** |
| 000000000 (= D) **2** | 000001000 (= D) → 000010000 (D << 1) **12** |
| aaabbc (From 7th to 12th, 9th a in Red )<br>111000000 (a, 9th)<br>001100000 (b << 1, 10th)<br>011000000 (b << 2, 11th)<br>011110000 (c << 3, 12th) **3** | 000010000<br>000110000 (b, 11th)<br>000010000 ( =D ) → 000100000 ( D << 1) **13**<br>000100000<br>000110000 ( B, 10th) **14** |
| 001000000 ( = D )<br>010000000 (D <<1 ) **4** | 000100000 ( =D ) → 001000000 ( D << 1) **15** |
| 010000000<br>111000000 (a , 8th) **5** | 001000000<br>111000000 (a, 9th) **16** |
| 010000000 (= D)<br>100000000 (D << 1) **6** | 001000000 ( = D ) → 010000000 ( D << 1) **17** |
| 100000000<br>111000000 (a , 7th) **7** | 010000000<br>111000000 ( a , 8th) **18** |
| 100000000 ( = D)<br>000000000 ( D<<1 ) **8** | 010000000 ( =D ) → 100000000 (D << 1) **19** |
| 000000000<br>000001111 (c, 6th) **9** | 100000000<br>111000000 (a, 7th) **20** |
| 000000000 ( =D ) so jump to the 15th place because so far 6 places are detected. **10** | 100000000 ( =D )<br>Hence a pattern is reported. **21** |

Figure 1.5: A complete worked example of SBNDM4

bit masks for this as following:
a = 111000000.
b = 000110000.
c = 000001111.

After mask generation, we will calculate jump size according to formula $m - q + 1 = 9 - 4 + 1 = 6$. After having processed that far we will begin to process the given text. Refer to figure 1.5 for delicate details. As the jump size is 6, we will jump directly to the character 6th character in the given text which is ’c’, and will begin to processes 4th character to the left of this which is again ’c’, as q = 4. This is depicted in portion labeled as bold 1.

From this ’c’, masks are aligned, with each successive character towards right with a single left shift (depicted in portion 1). After aligning we will take a bit-wise AND of all four characters. The result of this operation is stored in state register D which is also nine bit wide. As this time the result is all zero, so we will jump to position 12, which is character ’c’. This is depicted in portion 3 of figure 1.5.

We will start our operation from fourth character to the left of this ’c’, which is 9th, and is character a. From this ’a’, we will align all other three characters to the right of this till 12th character, which is ’c’. After AND operation is done the resulting value is 001000000 ( = D ), which is non zero. So we will continue scanning now towards left of character nine, but first we will shift the D to left one time, which will yield a value 010000000 (D $\ll$ 1 ). In portion 5 of figure 1.5, we will AND the 8th character, which is ’a’ with current value of D. The result is 010000000 (= D) and after a left shift 100000000 (D $\ll$ 1). This value is not zero, so will continue scanning. The next character is 7th, character ’a’. After AND with current value of D, the resultant is again non zero 100000000 ( = D). But after left shift it becomes all zero, 000000000 ( D $\ll$ 1 ).

In portion 8 we can see this. In portion 9 when we will AND current value of D with character 6th, which is character ’c’, then it results in all zero. So far we have searched the longest prefix of the pattern. So we will jump 3 characters further right to the 12th character, where we previously jumped. Now we are at 15th position(character ’c’). Now again we will follow the same pattern we followed earlier. As depicted in portion 11, right from character 12, which is ’c’, towards 15th character we will align all character with each one with successive shifts. The result is 000001000 (=D). And after left shift 000010000 (D $\ll$ 1) this is depicted in portion 12. Next character is ’b’, which is 11th. Again the result is non zero 000010000 (=D). After a left shift this becomes 000100000 ( D $\ll$ 1). This process will continue till character 7, and is depicted in portions 13 to 21. When a pattern is found it is found one position to the right of parameter j, which is *pos* parameter(see fig 1.4). So it is reported at position $j + 1$. The current pointer again jumps 9 + 6 characters for next scan to begin. The scan continue until the character ’$n - q + 1$’ is reached.

### 1.6.3 Summary

In this chapter we started with the some introduction to hardware accelerators. We discussed different types of accelerators. Then we discussed what are String Search Algorithms. We studied in detail what are DAWG or Suffix trees and how they are related to string searching problems. We explored what is bit-parallelism, and how it makes things work faster in string searching. Then we took a detailed look at SBNDMq mathematical background. In the last of this chapter, we did a detailed example fully outlining and understanding the concept of SBNDM4 algorithm.

In chapter 2 we will discuss some of reference research work already done and published. Our focus will be mainly on infrastructure, organization and resources used for undertaking a particular application work. In chapter 3 we will discuss the basic design of SBNDM4 processor in Verilog. In chapter 4 RAM design for our purpose will be discussed. We will see interfacing issues in detail. In chapter 5, we will discuss some additional parts of the our system and will combine all system parts, processor replication process will also be discuss.

# Chapter 2

# Literature review

*In this chapter we will see how FPGAs are used to accelerate different applications. We will see how hardware resources and computers are arranged to perform tasks faster. We will also see how string search has been implemented on FPGAs by different researchers. The main emphasis will be on the network or the setup used in different experiments.*

## 2.1 FPGA for Acceleration

As we know that there are two kinds of computer processes, ones that may be processed serially and others may be done in parallel[41]. FPGAs have their most use when it comes to the processes which may be done in parallel. The basic fabric of FPGA supports parallel processes, accumulated effect of which may surpass the performance of any system working at much higher frequency and processing data serially (typically microprocessor systems).

Until late 90's microprocessor based systems were the main choice for High Performance Computing. They were used as replacement of conventional supercomputer systems because they were cheap, and performance was scaling up with frequency and increasing silicon transistor density. But this touched its plateau as frequency cannot be increased beyond a limit due to increasing power consumption and certain signal characteristics. To keep up the pace, multicore systems were developed. But this again is facing a limit as multicore processing models push developers to adopt parallel programming to get most out of this methodology. But this again has limits. So this is where FPGA comes into play.

FPGAs are widely being used to accelerate applications right from servers to dedicated computers. In the past high performance computing community was hesitant to use FPGAs due to their higher costs. But with passage of time cost per FPGA has dropped significantly. They are now being widely used in numerous applications.

## 2.2 String Search Applications

### 2.2.1 Communication Network Paradigm

String search algorithms and methods are being extensively used in the field of network security and traffic monitoring. Network Intrusion protection is being performed by inspection of all incoming data into a network. We search for a particular pattern in computer commands or deep into packet contents. String searching algorithms have also their applications into other networks aspects, for example Network Direction Lookup Applications such as DNS, LDAP etc. Application involving high throughput or data intensive applications always prefer hardware based string searching. The hardware based string search system have also been proven useful in the field of Data mining e.g. Mercury System [42].

Network Intrusion Detection Systems usually are of two types: 1) Anomaly Detector Systems, 2) Signature Detection Systems. Signature detection systems usually compare network traffic with already very well known patterns. Such type of methodology has very low false alarm rates though, but it is not very effective against the attacks having new signatures.

With the changing demands of Intrusion Detection Systems(IDS), and with slower software based systems in hands, researchers began to explore reconfigurable systems to exploit parallelism offered by these platforms. In recent years many authors have implemented their work on FPGAs. The main emphasis of almost all research have been to accelerate the compute intensive processes. For example in [43, 44, 45] authors have implemented the TCP state-tracking and frame reformation. Some have done packet data deep inspection for malicious data detection [46, 47]. While these solutions offers a significant edge over the counterpart software implementations of IDS, they lack in addressing the data rate efficiency of the data being transferred between Network Interface and Intrusion Detection part.

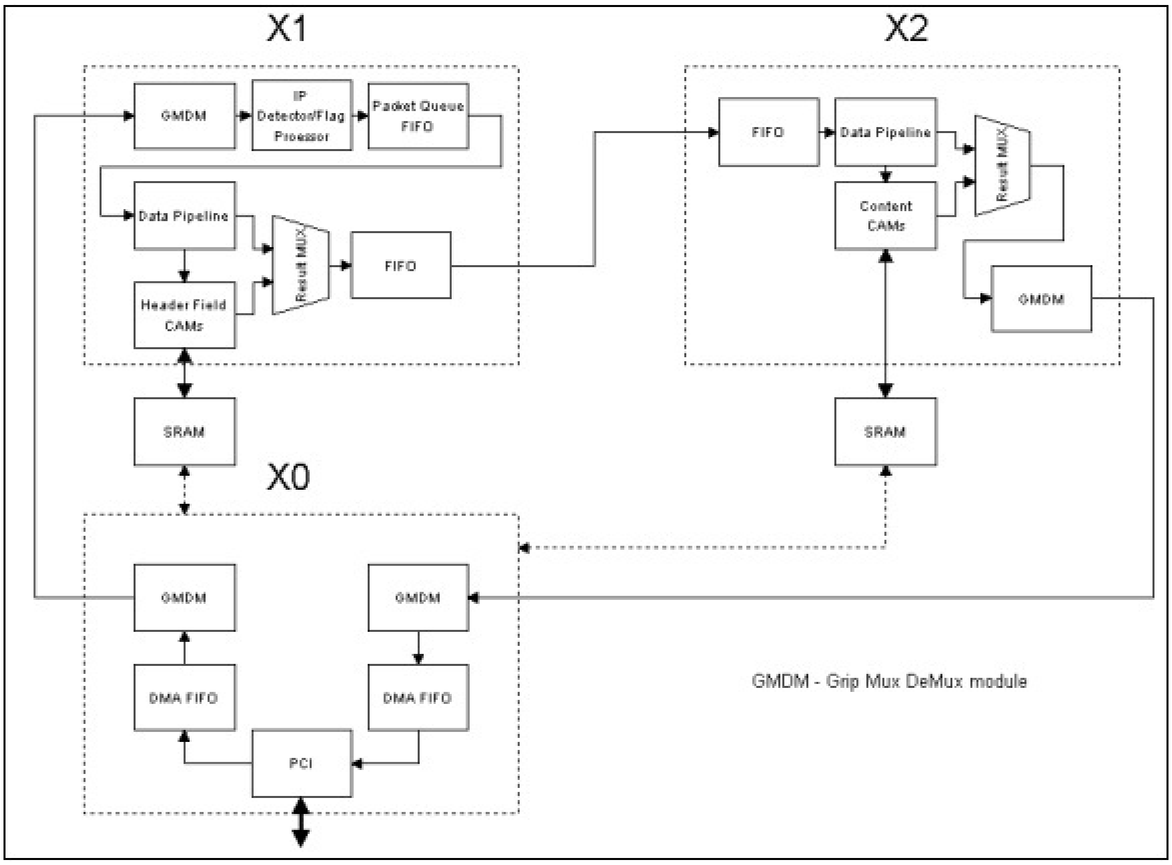


Figure 2.1: An IDS system example 1 [1]

There are some complete solutions in the literature we have so far described, that contain FPGAs. By complete solution we mean a product, capable of taking inputs from user, process queries and return the results. All of these solutions attempted to improve the bottlenecks of software based implementations. But these systems differ in the choice of system parts and functionality distribution among different parts.

[1]A courtesy by [48]

In [48] FPGA based IDS shares Intrusion Detection processing from a software based IDS system. In this approach an FPGA board is first connected to a daughter card handling to enable the board for PCI communication. The daughter card supported 1Gb/s. After connecting these cards , this setup was connected to a PC through PCI port. In this setup FPGA, which ran IDS system, scanned packets headers and payloads to match with certain existing rules, and then reported to the software running on the PC about the matching rules. Although the FPGA which is used in this experiment can process network packets at 2Gb/s, but the network interface becomes the bottleneck, limiting the data rate to 700Mb/s. The reference figure is 2.1.

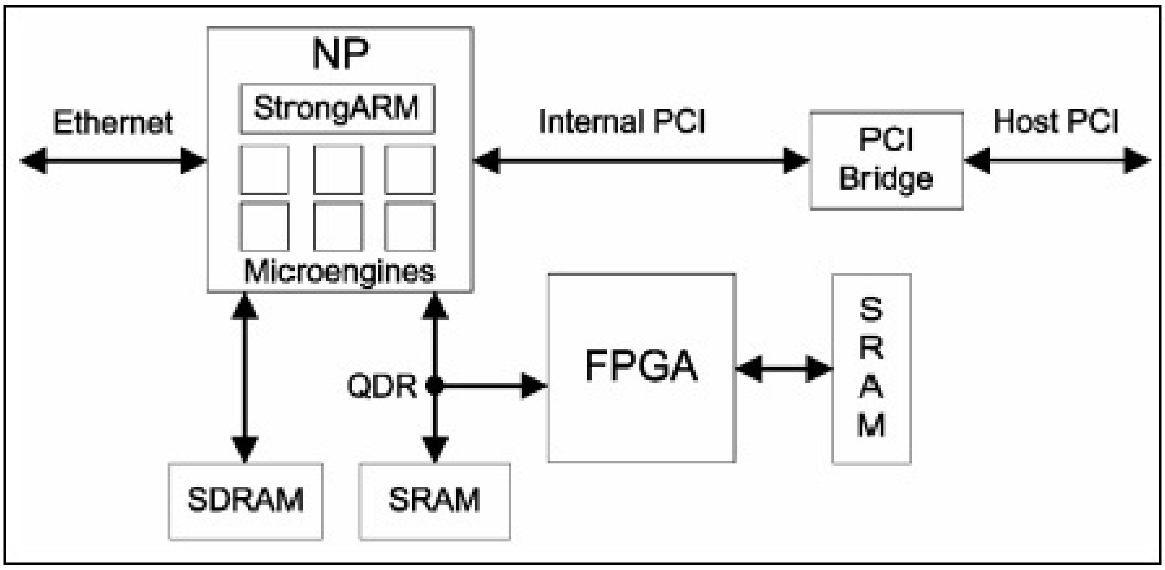


Figure 2.2: An IDS system example 2 [2]

A different approach has been proposed by Clark et al. in [49]. The purpose of their work is to provide whole the Network Intrusion detection on a network card. The design uses commercially available FPGA boards and Network cards. In this design Network processor card is connected between FPGA board and PC. The connection between network processors and PC is through PCI port. Similarly the connection between FPGA and Network processor is via PMC connector. Inclusion of software programmable network processors enables more complex processing of network packets before they are forwarded to host. The FPGA is used here as an accelerator for network pattern matching. In this proposed systems, bottle-neck is the communication between Network Processor and FPGA, as it reduces the through-put by 10% of what was achievable when design used FPGA only. The reference figure is 2.2.

In [50] Lockwood et al. proposes a firewall implementation. In this approach, the control is distributed among the network cards and FPGAs boards with which network cards are connected. On each FPGA board, one FPGA is dedicated to interact and communicate with the system, and other is dedicated to implement secondary network management systems such as IDS. All parts of this setup may run upto a speed of 2Gb/s.

[2]A courtesy by [49]

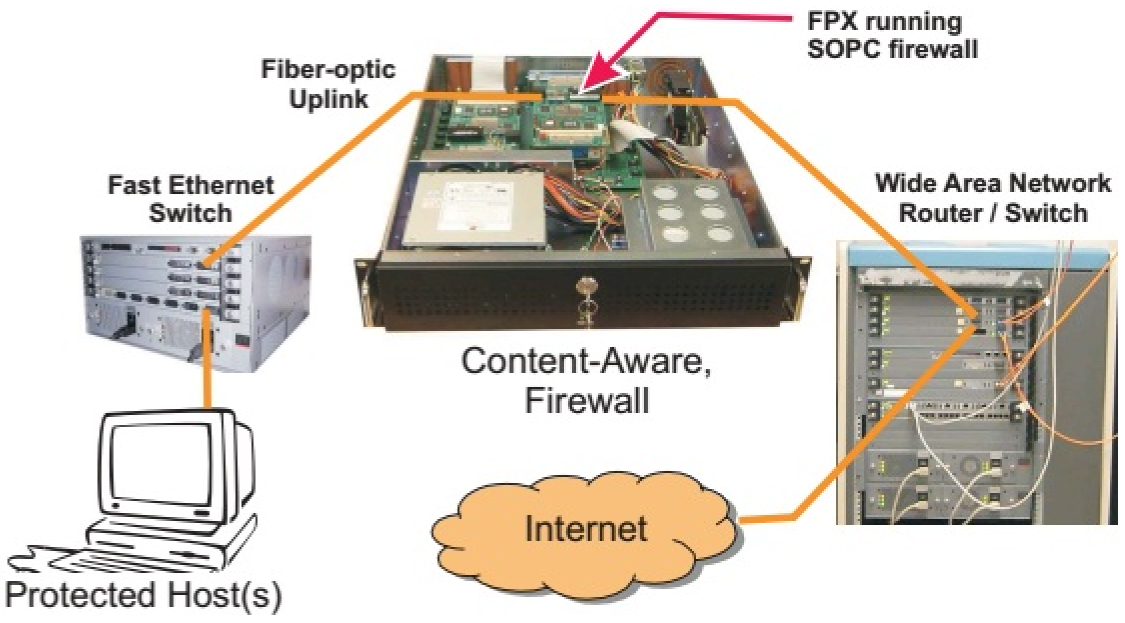


Figure 2.3: An IDS system example 3 [3]

### 2.2.2 Bioinformatics

With the passage of time the variety of genomic data available through different bio-technologies is increasing. DNA and Peptide sequences has been the main areas of focus throughout all recent modern times. Most of the main operations performed are to find any similarities with a given sample of data set. In the field of Bioinformatics, sequence matching is called “alignment". Conventionally, when a single request is generated, the main target is to report back all occurrences of a given pattern. Some other applications also involve the comparison of more than one patterns to be compared with complete given Genome-set. Either it is one target pattern or multiple, the focus of computation is always to find how many times this pattern(s) may be found in the given database or Genome-set[50].

Since 1980’s, the genomic data is expanding almost exponentially. For example, UniProtKB/TrEMBL database [52], that contains approximately 700 million sequence entries as off October, 2016. It contained 8 million entries in 2013 [50]. That shows a 12 times increase. In addition to that, recent advances in the analysis tools and their availability have added many new dynamics in genome research [53]. So, the daunting task of analyzing this flooding data, is becoming harder by each day with conventional schemes. These tasks becomes even more difficult with the keeping the plateau in the processor speeds.

A lot of work has been done on parallel data processing as when a huge amount of data is available, a straight solution is to divide or partition the data into smaller parts and process them separately and in parallel, and in the end combining the results. So in this paradigm, there are two types of schemes which are available. First are those involving main frames and cluster, where every single computer is assigned different data, and don’t have to keep the performance or process stage records of other computers. Second approach is based upon processing individual queries in parallel by processing units within a single processor. This fine grained parallelism may be achieved through either Field Programmable gate array or Graphical Processing Units. The biggest advantage of using a FPGA is lower cost at operation, power

[3] A courtesy by [50]

and size. This is also more faster, and today's high-end FPGAs may perform faster many times than that of a PC.

But these two approaches may also coexist, giving more advantage in terms of processing power and cost. Few nodes within a cluster may be fitted or connected with FPGA, GPU or a hybrid accelerator(s). Whenever data to be analyzed gets bigger, some data may be routed to these accelerators for load sharing purposes. There are usually two types of methodologies or algorithms adopted while implementing searching or aligning schemes. Ones are Dynamic Programming based algorithms and others being seed based algorithms. In Dynamic programming methods we employ two main attributes. First is the fact that aligning algorithms are lengthy, so processes needs to be accelerated. Other being is the fact that steps are well known, and really go well on highly parallel computation environment. Second type of algorithms are based on seeds. These algorithms reduce and limit the number of searchings, hence reduce the computations, as compare to Dynamic programming schemes. Two of the seed based algorithms are FASTA[54] and BLAST[55].

Now we will see some schemes implemented based on the algorithms we discussed in previous paragraphs. We will concentrate on FPGA based implementations. In [56] authors propose a Variant of BLAST algorithm with the name RC-BLAST.

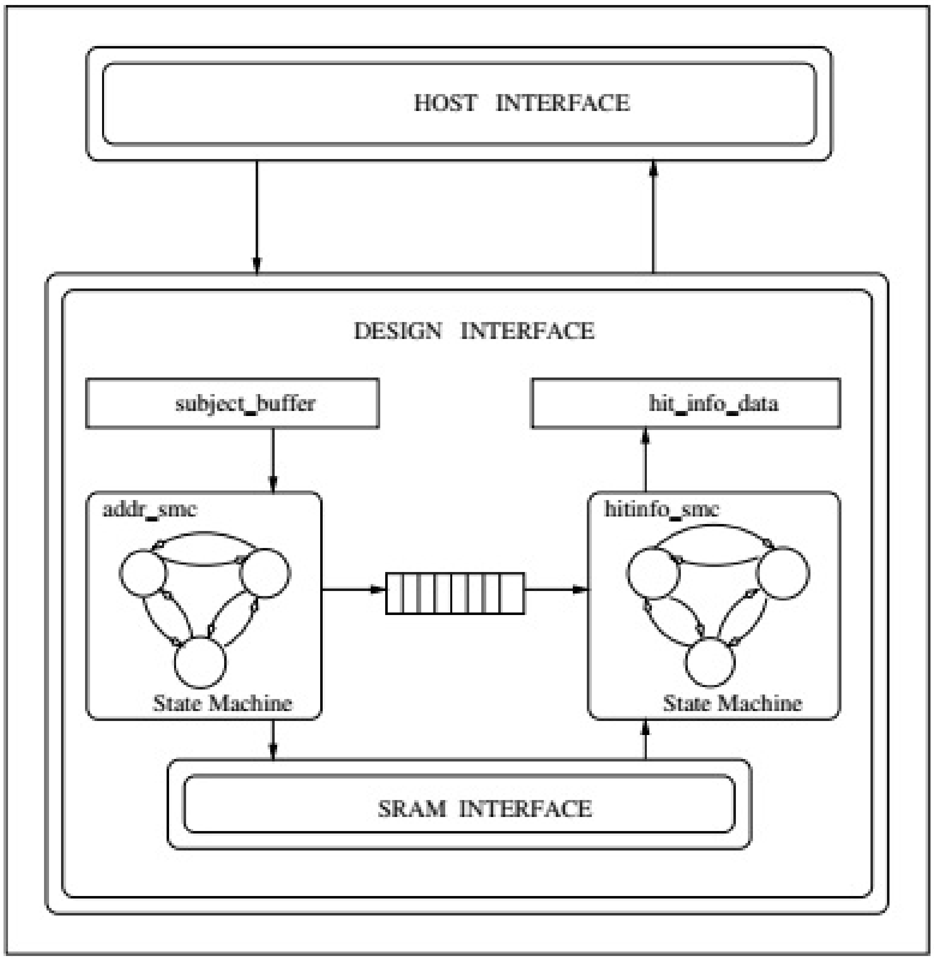


Figure 2.4: A RC-BLAST architecture [4]

[4]A courtesy by [56]

In this paper, BLAST is implemented on a re-configurable hardware(hence the name RC). This re-configurable hardware is obviously a FPGA. The FPGA used in this experiment is XC4085XLA by Xilinx. This paper presented a FPGA based BLAST algorithm accelerator. The accelerator is connected to a i386 machine, running RED-9.0-Linux operating system. The FPGA card used in the experiment contains two FPGA chips, with each one having 1MB of block RAM. The interface offered by the card is a PCI port. In their experiment they used a single FPGA and a single Block RAM set. Their experiment yielded that for BLAST implementation resource utilization was just around 25% of overall resources available on the target FPGA chip.

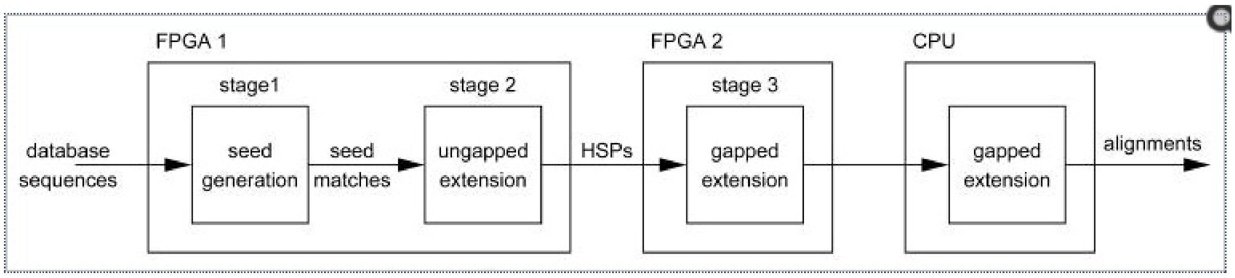


Figure 2.5: A Mercury-BLASTP architecture [5]

In [57] authors have presented an implementation of BLAST with a name Mercury-BLASTP. This paper utilizes a version of BLAST which is specially made for protein sample analysis(hence the suffix P). The FPGA used in this experiment is Virtex-2-6000 by Xilinx. The workstation used in this experiment is commodity PC. They have implemented their experiment in three stages: 1) Seed generation 2) Ungapped extension 3) Gapped extension and 4) Query support. The experiment setup comprises of two 2.4GHz, dual-core processors Opteron CPU, containing 16GB RAM, running 64 bit Linux. FPGA board is connected via a PCI bus to host. There two of such FPGA boards. First board implements first two stages of the algorithm. Second FPGA implements the third stage of the algorithm. The paper reports that this system performed 11-15 times faster.

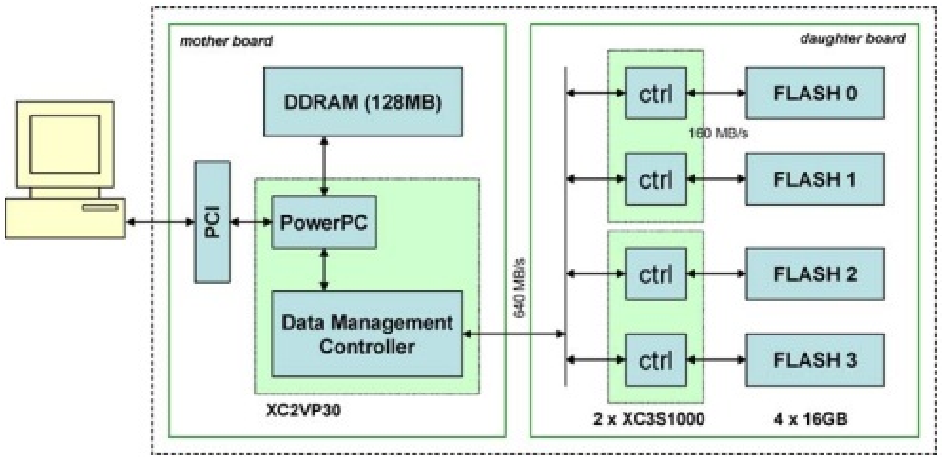


Figure 2.6: A RMEM architecture [6]

[5]A courtesy by [57]

[6]A courtesy by [58]

In [58], an acceleration application has been introduced that utilizes Flash memory and FPGA for speeding-up Genome processing through using Seed based algorithm. In this experiment a commercially available Flash memory board was plugged into an FPGA board. Resulting board was exploited for two different applications: database search and intensive comparisons. Database search application runs with significant accuracy on RMEM setup, and performs better than a dedicated database search server. The setup shows more promising results against shorts strings. This betterment in performance is mainly due to the availability of data in Flash memory in the vicinity of FPGA processor. The setup for this experiment is depicted in figure 2.6.

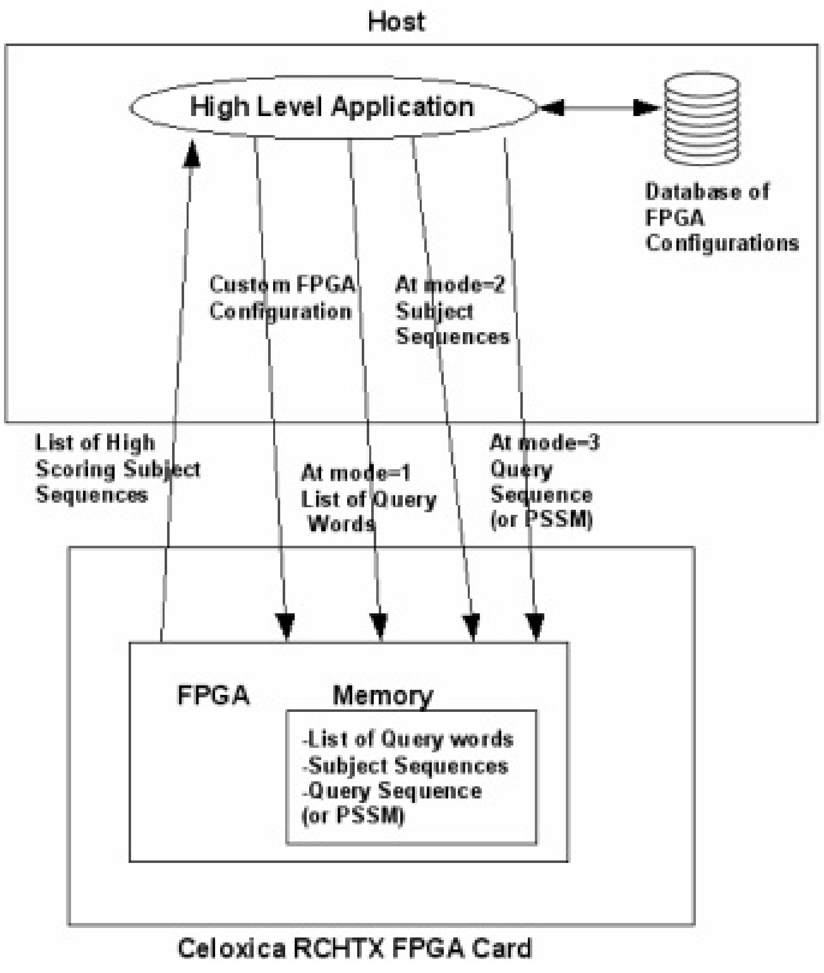

Host
High Level Application
Database of FPGA Configurations
Custom FPGA Configuration
At mode=2 Subject Sequences
List of High Scoring Subject Sequences
At mode=1 List of Query Words
At mode=3 Query Sequence (or PSSM)
FPGA
Memory
-List of Query words
-Subject Sequences
-Query Sequence
(or PSSM)
Celoxica RCHTX FPGA Card

Figure 2.7: A FPGA based PSI-BLAST accelerator [7]

In [59] first ever implementation of PSI-BLAST is presented. In this experiment authors have used XC4VLX160 FPGA by Xilinx. The IP core was created in C-programming, and after generating bitstream, downloaded to target FPGA. Only Block memory was used in the design, and Swiss-Prot protein sequence database was the reference for this experiment. The hardware results are compared with a C program implementation on Intel Centrino Duo work station having 2.2GHz processors with 2GB main memory. FPGA accelerator, while clocked at just 15MHz, performed at 20 to 40 times better than pure software based implementation.

[7]A courtesy by [59]

## 2.3 Other applications

In the following section we will examine some of the research literature, that may be beneficial for our work. As previously, our emphasis will be mainly on experiment setup.

### 2.3.1 Compute Intensive Applications

In [60], a PC-FPGA architecture for Floating Point matrix multiplication is demonstrated as depicted in figure 2.8.

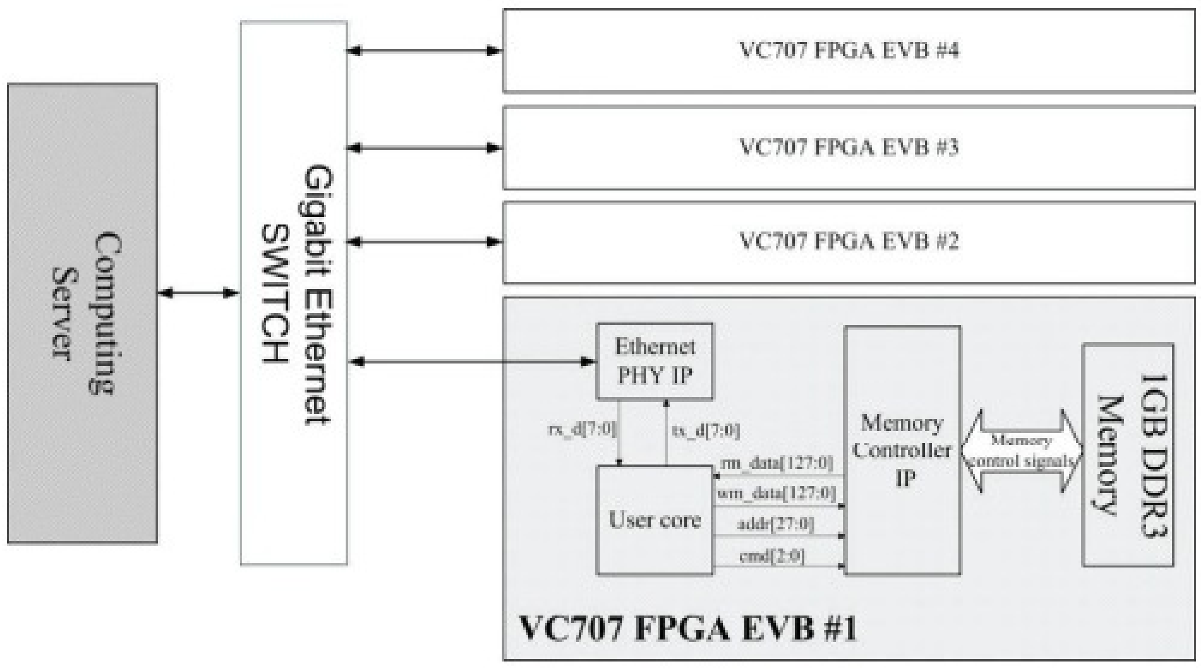


Figure 2.8: A Setup for Floating Point Matrix Multiplication [8]

Computers perform their operation on Floating Point(FP) numbers rather than normal decimal point numbers or integers. These operations are highly compute intensive, and even multiplication of two matrices of moderate dimensions can cause computers to take significant amount of time. This is because of inherent architecture of conventional microprocessor based systems. On the other hand, FPGA fabrics support massive parallelism due to their fine grain methodology. In [60] authors have combined two commercially available processors for big data computations. The FPGA used in the experiment is Vitex-7, and evaluation boards of Xilinx FPGA are used. These evaluation boards are connected to a Gigabit Ethernet Switch through Ethernet port, and the Switch is connected to a server computer. Each evaluation board is running at 125MHz clock frequency. The results are compared with *i*7 processor based CPU, running at 3.4GHz, and matrix multiplication speed-up of 4x is observed. In this experiment data is first downloaded to DDR2 from PC. DD2 is available on the board. Data is then retrieved for performing calculations. Similarly, results are also written back in the similar manner from FPGA to DD2 and from DD2 to PC. In [61] a similar approach has been adopted in which authors have proposed a hardware with the name RC100. This is a hybrid system

[8]A courtesy by [60]

of FPGA and ASIC. In [62] a very elaborated scheme is presented(depicted in figure 2.9). In this approach authors have presented a cluster made of PCs. This cluster is intended for matrix multiplications. Each node in the cluster is connected with an FPGA based accelerator(refer fig 2.9). The basic communication between node and FPGA boards(accelerator) is through PCI port. Whereas the interface for communication between nodes and server is a Message Passing Interface(MPI).

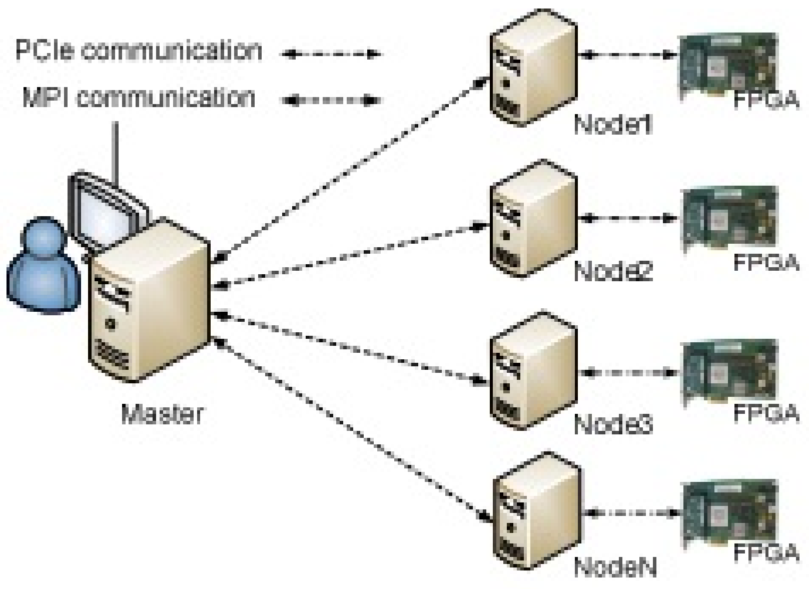


Figure 2.9: Massive FP matrix multiplier [9]

The FPGA used for the accelerator are VLX50T. In this paper, a query is generated from main server(master) ,which is passed down to FPGA accelerator for computations. The data is first loaded within DDR2, available on FPGA boards, and then called into processor for further processing. The performance of this setup is compared with software implementation running on Intel Core2-Quad-Q8400. Results show that the speedup was 1.19 times.

Similarly other such accelerators may also be viewed in [63, 64, 65].

## 2.3.2 Applications with SATA interface

SATA stands for Serial AT Attachment. This is an interface standard, devised for many modern communication applications. Most of the modern massive storage devices, including Hard drives for PCs have SATA interface. The reason why this interface is important for us will be discussed in section 2.4. As for data transfer rates , SATA is reaching 16Gbps. This much speed combined with massive parallelism offered by FPGA may be beneficial for many applications. Let's see few examples, involving String Search, FPGA and SATA interface.

In [66] authors have proposed a design (most probably for commercial purposes) that provides a FPGA-mass storage devices solution in form of boards. The said boards offers many Gigabit Ethernet and PCI ports, a single FPGA on each board, with each FPGA capable of handling multi-gigabit communication and switching simultaneously. The brain(an FPGA) of the architecture is a Xilinx Spartan-6 chip. This chip contains PCI blocks and supports data rates up to 3.2Gb/s. This platform also incorporates 128MB of DDR3 memory. This system at the Ethernet side provides supports for Reduced-GMII interface. For communication with SATA

[9]A courtesy by [62]

devices, authors have used GTP blocks available on the FPGA chip for communication with host PCI bus. When two such cards are connected in master slave configuration, connected via Ethernet bus, can sustain a data rate of 118Mb/s. And as always the data rates of the system are limited by interface and in this case it is Ethernet. Such type of setups are very useful for the applications requiring large data search. SATA based Mass Storage Devices in vicinity add value to this purpose, as the capacity of commercially available devices is increasing exponentially.

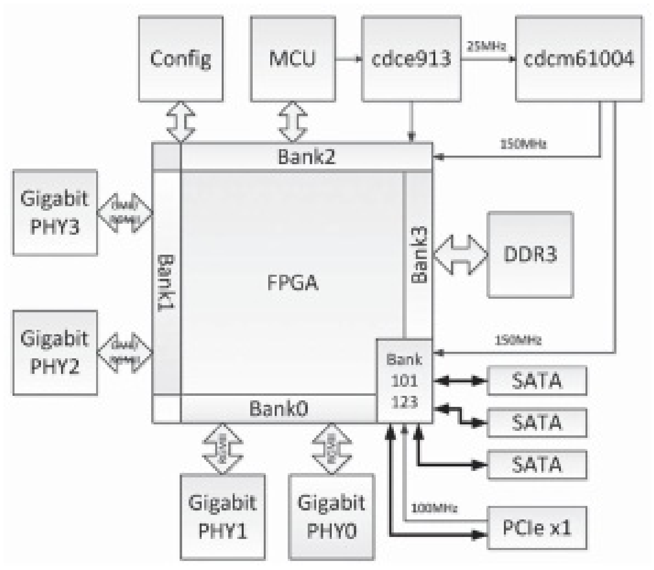


Figure 2.10: A NetStorageFPGA system overview [10]

In [67] (as also depicted in figure 2.11) authors present an architecture that is useful for DIALIGN algorithm implementation.

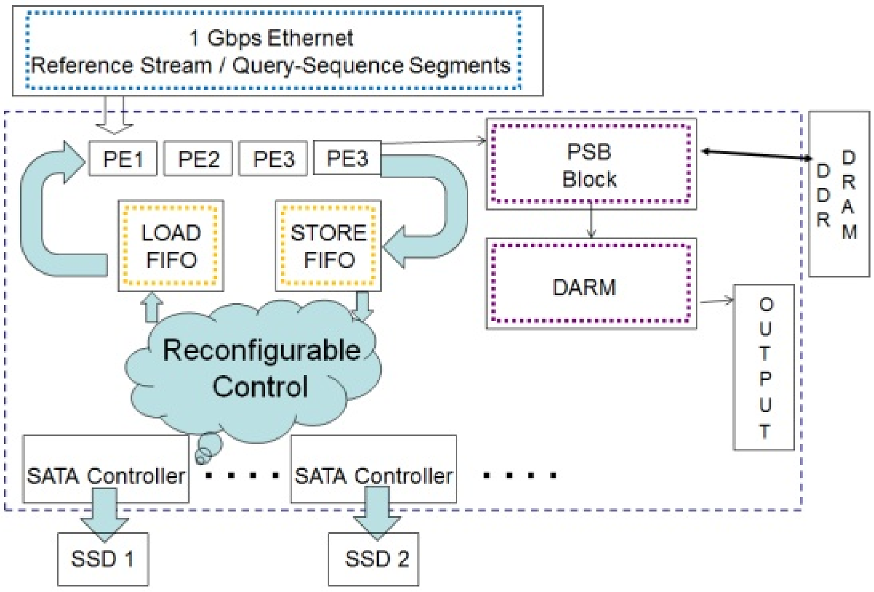


Figure 2.11: DIALIGN algorithm alignment platform [11]

[10]A courtesy by [66]
[11]A courtesy by [67]

In genome alignment (pattern matching) processes, reference pattern length can't exceed a certain limit due to configurable or manageable dimensions of underlying computational structure. This paper deals with the same problem by increasing the target pattern length to Mega and Giga bit long. The basic method of operation is to stream reference or target pattern to FPGA-based aligner, that compares this stream with a database available in SATA SSDs(Solid State Devices). In this paper, authors have implemented DIALIGN algorithm on Xilinx Vitex-5 FPGA, and the target board is XUPV5-LX110T development board. This board also features two SATA connectors and a Gigabit Ethernet connector. Essentially this system consumes 80% of the target FPGA. The maximum operating clock rate is approximately 67MHz. In the experiment a 200 base-pair query is generated by the computer against a 1GB reference data. It took almost 29 mins to complete the alignment(matching).

## 2.4 Motivation

SBNDM4 belongs to a class of String Searching Algorithms, which fall under Exact String Search Algorithms. These algorithms find their applications in numerous fields like Bioinformatics, financial screening, web search engines, text search to software engineering. As we saw in literature review that approximate string search algorithm like BLAST and DIALIGN are implemented on FPGA within numerous setups [67, 56, 56]. We also reviewed some setups which support database search [59, 67]. Our motivation behind this thesis is to produce a research work that may pave the path for many other scientific researches and product designing. So for doing this, we have used the parts, programming language and methodologies in this work which are readily available and are easily implementable. Although the work is in its initial stages of development, it is scalable, modifiable and don't requires sophisticated apparatus or equipment for further development and analysis. SBNDMq algorithms are previously explored only by their implementations in software by researcher. However, our work will be some of initial works which has applied this algorithm in reconfigurable computing platform. During our work, we will put our most effort on specially the following aspects:

- The design should be as fast as possible, so that it is not causing any throughput bottlenecks.
- The design should occupy minimum space on chip, so that interfacing SBNDM4 scanning system's IP core with other communication and processing cores is easy.
- The interface will be kept as generalized as possible to keep the interfacing options open.
- The work will be done in a language that should be one of the industry standard.
- The design should be in a generic format for FPGAs, so that design may easily be migrated to other FPGAs as well.

We will conclude on a design which, with a slight modification, will be fully capable of working in any one of the configuration discussed in [48, 49, 56, 58, 60, 62, 67].

# Chapter 3

# The SBNDMq processor design

*In this chapter we will see how SBNDM4 algorithm works. We will look at detailed design of a SBNDMq processor and design issues. We will see a Verilog implementation and how this design was verified with the detailed examination of test benches and simulation results. Further, detailed overview of optimization techniques which we have applied for use of the same will be presented.*

As we have seen in section 1.3 that how SBNDM4 works. However from programming and hardware point of view, we must draw some definitive boundaries in order to realize SBNDM4 algorithm in hardware form. For doing this, let's assume that *m* is the pattern length, which is 9 in our case, and q is taken from SBNDM4 as 4. Then skip size may be calculated as: $(m - q + 1) = 9 - 4 + 1 = 6$. Mask generation is a first process in this algorithm and is termed as "*pre-processing*". We have assumed that data coming from Real Time Manager (RTM), will be data masked already to reduce this processing. We will see RTM in chapter 6. Similarly, this approach also reduces or essentially filters out those characters which are not the part of our target pattern. Although we have chosen the alphabet size to be 4 here, it may be chosen right from 1 to 9, thus covering most of the important applications like Peptide Search or Alphabet Search(up to nine character). We take an example pattern as *aabbccddd*. The following masks for individual characters will be generated after pre-processing:

- *a* = 110000000
- *b* = 001100000
- *c* = 000011000
- *b* = 000000111

These masks will be stored in individual unique addresses within block RAMs, and will be called by the SBNDM4 processor, by corresponding address presented on address input. Each character will be read from BRAM in exactly one clock cycle. This very same character will be processed upon next negative edge.

It may further be noted if a character, while being a part of alphabet, but not the part of pattern, then such character will be assigned all zero or 000000000 mask. Here comes the point where we suggested that the pre-processing may be done by RTM. By doing so, RTM to end node traffic may be minimized. An algorithm may be implemented on RTM side to replace any consecutive(more than one) such characters by just a single such character. This will indicate a break at that point, hence eliminating the chance of false pattern detection, as skipping all zero pattern may combine two non-zero patterns.

After freezing these specs for hardware implementation, now we will discuss the design of SBNDM4 processor in Verilog HDL.

## 3.1 Block Diagram of SBNDM4 Processor

We have designed the processor keeping following objectives in our mind:

- The processor should accept a global reset.
- The processor will drive the Block RAM, that means it will generate addresses, and will accept outputs of RAM (individual characters).
- It will report back the finding of the pattern. Address of the locations from where the pattern is starting will be reported.
- It will report the completion of scanning process.
- It will generate a busy signal during scanning process of BRAMs to prevent overwriting of RAM contents.
- It will operate on system clock.

A block diagram of SBNDM4 processor may be seen below:

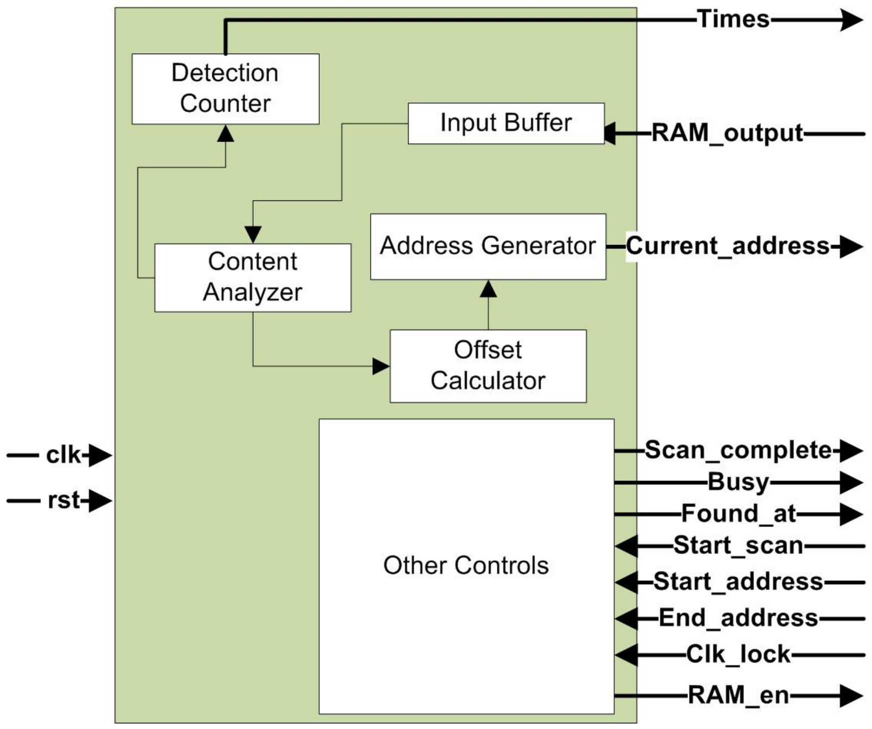


Figure 3.1: SBNDM4 processor block diagram

## 3.2 SBNDM4 processor: Details of operations

The module is designed in Verilog. At the start of design process, we had two choices for data buffering, coming from Run Time Manager(RTM), 1) either we store the data in buffers made using distributed resources of FPGA e.g. FFs, available within FPGA Slices, which is very much resource consuming, or 2) we are using existing BRAMs available within FPGA fabric, which is very much resource efficient. So we decided to use BRAMs.

There are 60 BRAMs in Virtex-5 fabric, each with 36kb space. For most efficient usage, an aspect ratio of 36x1k, 18x2k, 9x4k is recommended[11]. We considered these settings for generating core for configuring BRAM. For our purpose, we used 9x4K BRAM configurations, which mean width of BRAM will be configured as 9(bits) and depth will be 4k(addresses). As there are 60 BRAMs, we decided to couple each processor with two of such BRAMs, making 9x8k BRAM for a single processor to scan. The decision is made on the basis of target design speed and available resources. We will see the details of this in chapter 5.

Now we will analyze the details of operation of this processor. After data write-up from main controller (Chapter 5) to BRAM, the processor waits for two signals to start it's operation. In figure 3.1 both the signal can be seen as *clk_lock* and *Start_scan*. *clk_lock* signal is received from another module generating clock at 3x rate than that of the system clock. We will see the details of that in chapter 5.

Upon reception of *clk_lock* or *Start_scan* it will wait for other till both are asserted high. Waiting for *clk_lock* is necessary to ensure that we are only starting operation when every part of the system is ready. Here lets first see the dimensions of different signals at input or output in table 3.1 .

| **Signal** | **Bit_Width** |
|---|---|
| **Inputs** | |
| clk | 1 |
| rst | 1 |
| Start_address | 13 |
| Start_scan | 1 |
| End_address | 13 |
| RAM_output | 9 |
| Clock_lock | 1 |
| **Outputs** | |
| Busy | 1 |
| RAM_en | 1 |
| Times | 10 |
| Scan_complete | 1 |
| Current_address | 13 |
| Found_at | 13 |

Table 3.1: SBNDM4 processor inputs and outputs

### 3.2.1 SBNDM4 Processor Inputs

From the table 3.1, *clk* signal is system clock. We have assumed and have designed our processor so that it is accepting clock generated by our reference board. An Oscillator of 100MHz is available on the target board. However, as we will see in chapter 5 our designs maximum speed of operation is 50MHz, so we will use a clock divider as clock source. *rst* signal is reset signal. This reset may be asserted from main controller. When this signal is asserted, all output signals will assume their reset states. *Start_address* and *End_address* represents start and end address values of BRAM, to be scanned by processor, hard coded in our case. These both values are 13 bit wide, representing an address space of 8K. We have combined two 36kb BRAMs, forming a total of 72kb of BRAM. Every single word is 9 bit wide. Thus we have a total of 8192 addresses ($0 - 8191$). *Start_scan* is a signal controlled by main controller. Upon assertion, processor begins to scan 8K BRAM. *RAM_output* is the output of BRAM, which appears at input of processor due to the address provided by the processor itself. The width this input is 9 bit. *Clock_lock* is a signal generated by DCM. This signal indicates that every block in the whole system is ready for operation. Its width is 1 bit. This signal is one of the two signals necessary for each processor to start operation.

### 3.2.2 SBNDM4 Processor Outputs

*Busy* signal is an indication for the rest of the system that processor is busy. In this state, system will be considered busy, and this signal travels back to RTM, which decides on the basis of this signal, where to route the next data available. The reset state of busy signal is logic low. *RAM_en* signal is asserted high simultaneously with the busy signal. This enable the respective BRAM for data read mode. This signal is 1 bit wide. Its reset state is logic low. *Times* is the signal which indicates that how many times the target pattern is found during scanning. This is 10 bits wide, hence covering upto a range of 1k patterns. It's reset state is all zeros. *Scan_complete* signal is an indication by the processor that target address space has been scanned, and the results are ready. Upon reception of this signal, an indication will be sent to RTM that results are ready. This signal is 1 bit wide, and its reset state is logic low. *Current_address* is a incremental output of the processor. This provides BRAM with address during negative clock cycle, and during next positive edge of the clock, data is available as per provided address at the output of BRAM. The width of this signal is 13 bits, may cover from address 0(base d) to 8191(base d). The reset state of this signal is all zero or zeroth address. *Found_at* is a signal for another circuitry in the system, reporting address occurrence of strings found in BRAM. Every time a pattern is found, its starting address is reported. This is 13 bit wide, and its reset state is all zeros.

### 3.2.3 Operation sequencing

*clk* signal is always in toggling position. *rst* can be asserted at any time, but it is assumed that this will be asserted when all results are sent back to RTM.

First part of overall activity is filling BRAM by main controller. Main controller doesn't attempt to fill BRAM until it senses a high *busy* signal. It again starts to fill BRAM as long as it senses the *busy* signal going low.

As long as the controller finishes filling BRAM with 9 bit words, it issues a *Start_scan* command to Processor. Upon assertion of *Start_scan*, Processor asserts *RAM_en* and *busy* outputs to BRAM and main controller respectively. It issues 0th address on *Current_address* output to BRAM address lines. All this takes place during negative edge of the clock cycle. During positive edge, data is available at the input of processor with the input name *RAM_output*. Processor takes inputs one by one, sequentially, while making decisions. Whenever a pattern is found, *Times* output is incremented and the location is updated at *Found_at* output.

## 3.3 Verilog code for SBNDM4 Processor

Please see Appendix A for further code details.

## 3.4 Verification through test benches

Functional Verification is done within Xilinx ISE software.
We kept timescale 10ns/1ps, which is equivalent to 50MHz. Waveform are shown below:

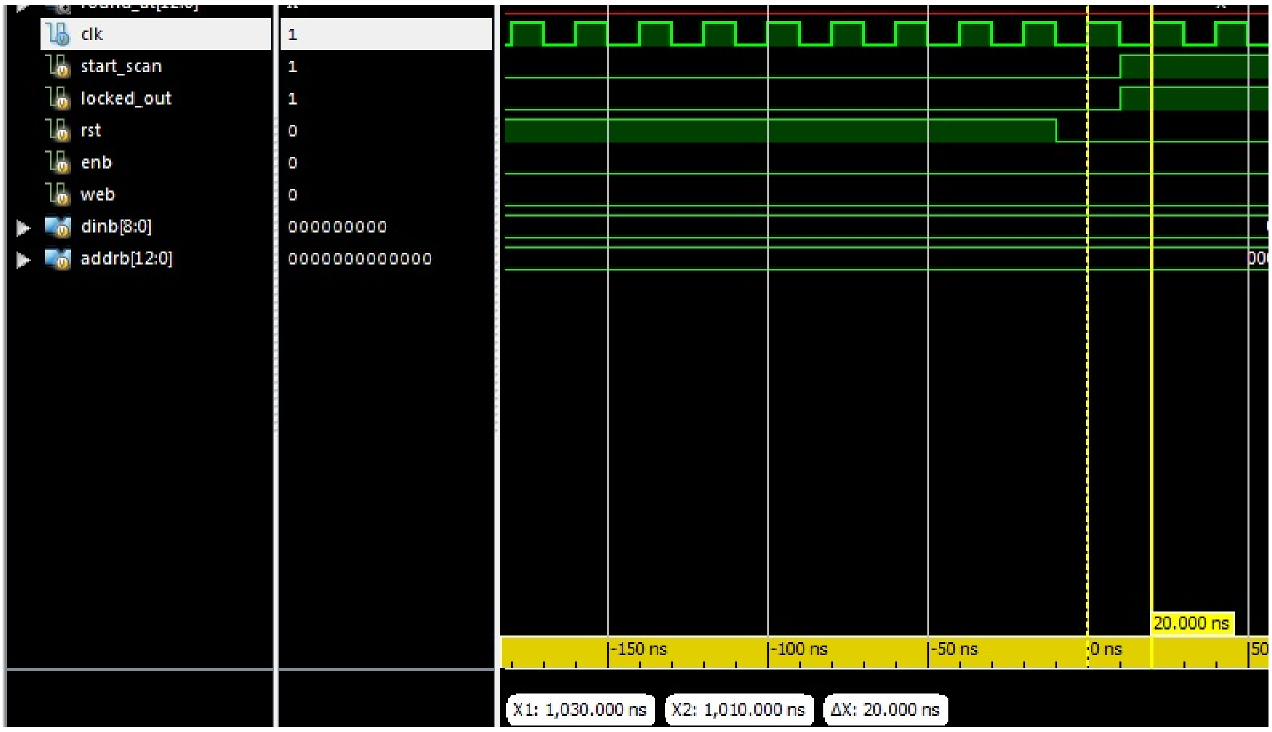


Figure 3.2: Clock generation for SBNDM4 Processor test bench

Processor is provided with both the signals *Start_scan* and *Clock_lock* at 1000ns (see fig 3.3). That triggered processors to start scan BRAM. As we already know that we are working to find out a pattern of 9 characters. The very same circuit may be configured to find out patterns of lengths 1 to 9. Now, we also know that we have initialized BRAM with known sequences of characters in order to test against known results. It may also be noted that in our case, we have initialized BRAM with back-to-back patterns, which mean patterns are one after other till the end, and processor can not skip a single character. So the timing obtained from these experiments are worst case, which mean, no matter what, processor will always either obtain better or the same results. So by that calculation, the said BRAM must contain 910 such patterns.

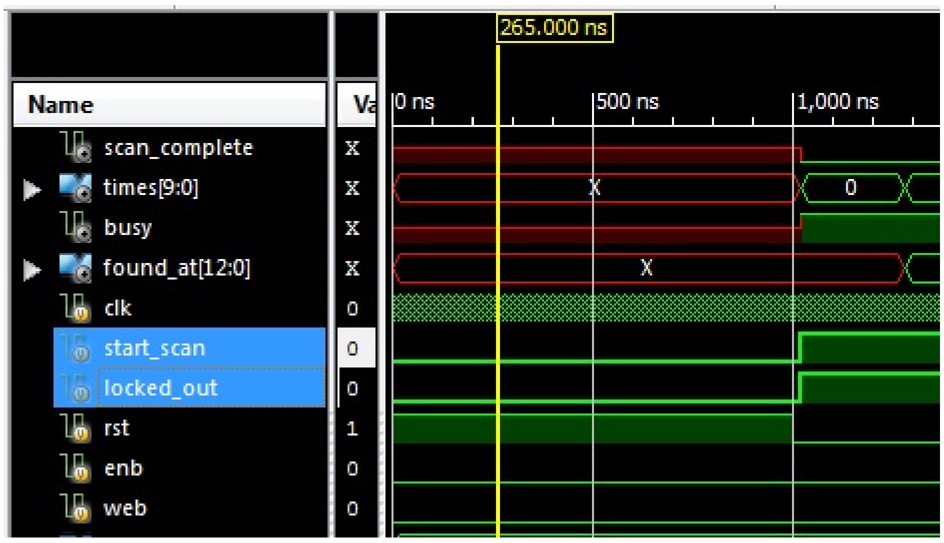


Figure 3.3: Application of *Start_scan* and *Clock_lock* signals

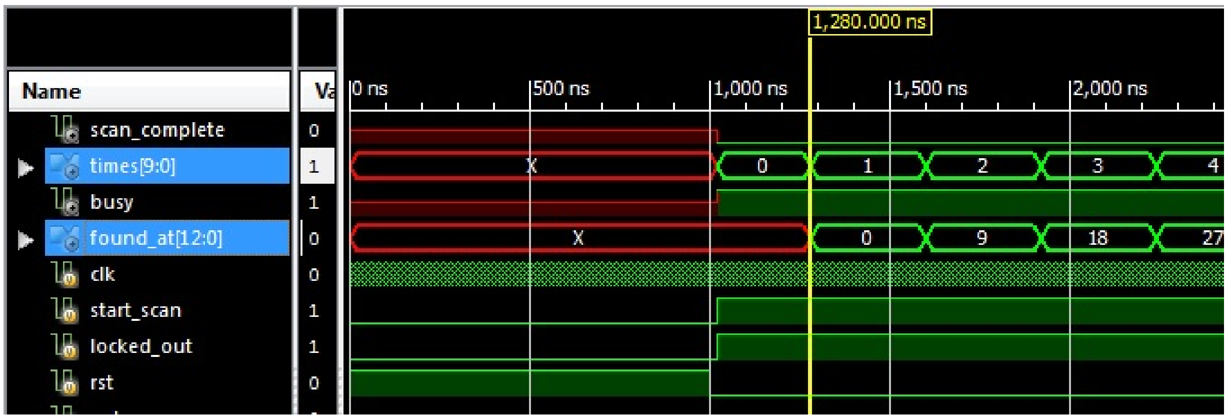


Figure 3.4: Pattern detection and where it was found

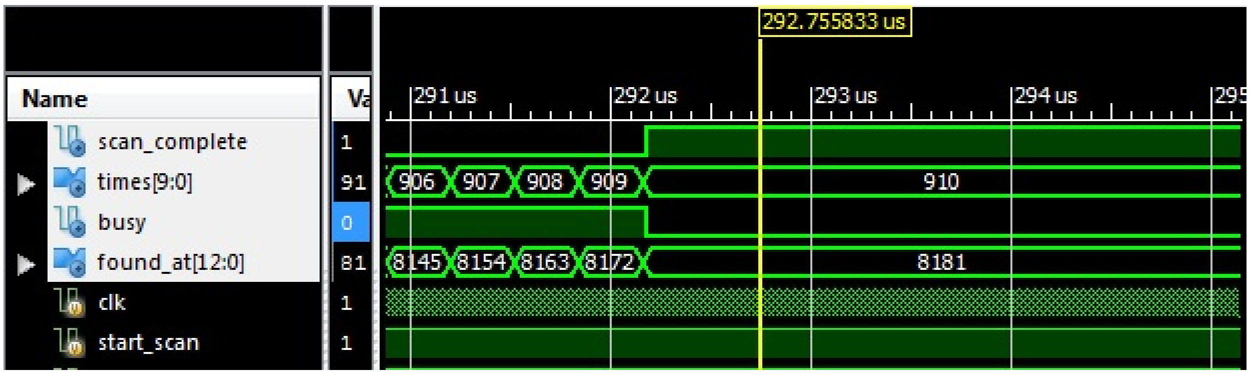


Figure 3.5: SBNDM4 processor outputs

First pattern is found at zeroth position (see fig 3.4), similarly second pattern is found at 9th position. In fig 3.4 it may be seen that busy signal is continuously asserted high. This remains high as long as Processor starts scanning and until last. Similarly *scan_complete* is asserted low throughout scanning process until last character gets scanned. Now we will analyze the outputs (see fig 3.5). It may be seen last pattern(910th) is found , the *Scan_complete* signal is asserted high, and at the same time *busy* signal is asserted low. This sends a message to RTM that results are ready, and circuit is free for next operation cycle to start.

## 3.5 Summery and results

We created an FPGA based SBNDM4 Processor in Verilog HDL language. This Processor is made to scan an adjacent BRAM of 8192 words capacity, where each word is 9 bit wide. The BRAMs are first filled or are written by main controller and then SBNDM4 Processor is given a signal to start scanning. Processor checks the whole BRAM for target pattern. During scanning, Processor keeps reporting the places(RAM addresses) where the pattern was found. After scanning is completed, the processor reports total number of patterns found. We will discuss the use of these signals, and control of other signals in next chapters. Processor takes 292us or 292,200ns to scan all the BRAM of 8192 words, and to assert the final status of *busy* and *Scan_complete* signals.

# Chapter 4

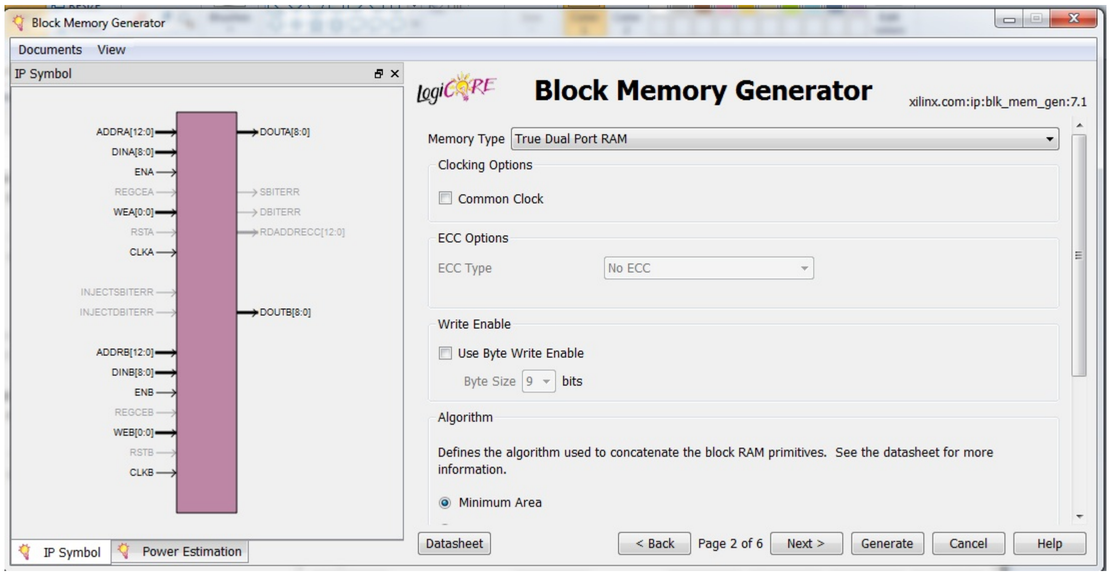
# Interfacing with Block RAM and Processor Replication

*In this chapter we will see interfacing issues of a Block-RAM(s) in our design. Multi-instance SBNDM4 will be discussed. Finally we will look into 24 processors instantiation, Block-RAM blocks, their interfacing and their parallel operation. Remedy for borderline cases will further be discussed.*

## 4.1 Block RAM generation through Coregen

For our purpose, we needed a RAM that may be read or written from two sides i.e. both the processor and main controller are able to access it on their turns.

So for this purpose, dual port RAM is good. Main issue with dual port RAM is that when same address for reading and writing ports appear, there will be a conflict. Now, as long as write first option is concerned, first the RAM will be written, and then the output will appear. To resolve this issue, we changed the input outputs of RAM in such a way that conflict scenario is removed now, no matter what. We will discuss that later, but first let's discuss RAM generation through Xilinx Core generator.



Block Memory Generator xilinx.com:ip:blk_mem_gen:7.1

Memory Type: True Dual Port RAM

Clocking Options
☐ Common Clock

ECC Options
ECC Type: No ECC

Write Enable
☐ Use Byte Write Enable
Byte Size: 9 bits

Algorithm
Defines the algorithm used to concatenate the block RAM primitives. See the datasheet for more information.
● Minimum Area

Figure 4.1: Dual port RAM generation through Xilinx Coregen

Long before generating this RAM, the space estimation was with us, so based on that calculation we generated our true dual port RAM based on these calculations (fig 4.1). At the very first page appeared after launching Core-generator within Xilinx ISE design suit, we selected the option of **True Dual Port RAM**. On the same page we also choose to algorithm as **minimum area**. Upon pressing **next** button, we choose Read and Write dimensions. We choose Write and Read widths as 9, as our pattern is 9 character wide and we are mapping each character using 9 bits. Similarly, Read and Write depths are chosen to be 8192(two 36k BRAMs). We choose these setting for both the ports (A and B). Similarly we also choose **write first** option and added **enable** option for both the ports. On next page we initialized RAM with a .coe file, which we made separately. We will discuss the contents of this file later in this chapter. For next all steps, we let the things as they were, and in last we chose to generate the core of a BRAM, that we will include in our design, along with HDL.

## 4.2 A Unique true dual port RAM

As discussed in section 4.1, we face a conflict situation when one of the port tries to write and other port tries to read from the same address. We adopted a simple remedy for this as depicted in figure below.

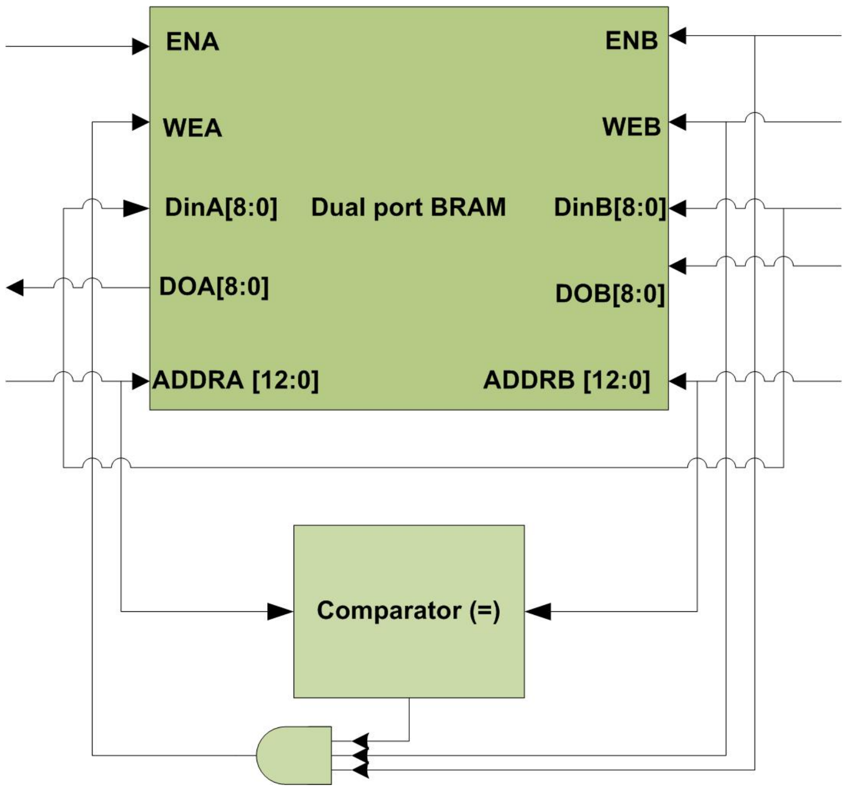


Figure 4.2: A conflict free dual port RAM

In the figure above ports A and B are shown. In this arrangement of RAM only one port i.e. DinB can be used to write in the RAM. The very same input is also connected to DinA. Now we will check whenever there is same address on both the address line (DinA and DinB). In that case comparator will yield a logic high output. Now besides that, we also look for ENB and WEB inputs, that means , whenever ENB and WEB is asserted high, and addresses are also same, it is highly likely that conflict will occur. So, we feed ENB and WEB to an AND logic along with comparator's output, and feed the output of the AND logic to WEA, so that whenever all these inputs are asserted like that, WEA is activated, and only write operation is performed. Data can only be read through DOA port, when an address on ADDRA is provided. Similarly data can only be written through DinB port when an address is applied on ADDRB. ENA and ENB will be driven from processor and main controller sides respectively.

## 4.3 A Processor and RAM block

Now we will discuss how SBNDM4 processor and this BRAM are interfaced.

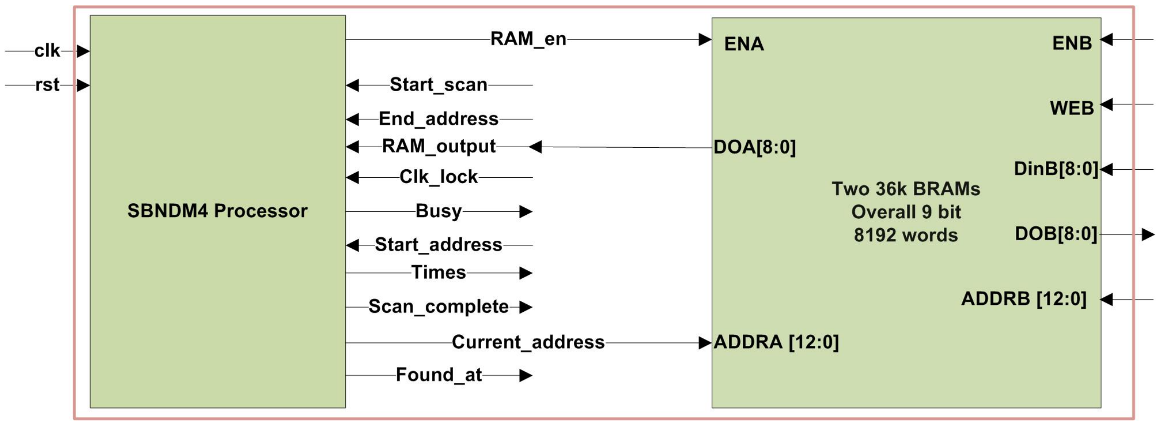


Figure 4.3: SBNDM4 processor and BRAM interface

In the figure 4.3, it may be clearly seen that how a processor-RAM block is formed.

SBNDM4 processor henceforth will referred as processor only. Processor has many outputs and inputs, only three out of them are meant to get connected with BRAM(henceforth referred to as RAM only). *clk* signal is common to all the system. When a *Start_scan* signal is asserted to processor, it issue an enable command to RAM, and ENA signal of RAM is asserted. Next comes the *Current_address* signal, which carries the address of the word, to be retrieved from RAM, and hence before the next positive clock cycle the RAM is provided with address on ADDRA line. On positive edge, data is available on output, from DOA line of RAM to *RAM_output* input of the processor. During negative clock cycle, decision about current input character is made. Next address is also generated during same negative edge and is presented on the ADDRA input of RAM and so forth. It may be noted that two circuitry are working on different clock edges. This technique is adopted to gain maximum speed.

As we already know there are 60 of such 36k bit RAMs available with our target FPGA chip. This block will consume two of them. It may be noted that we are not using simple RAMs but specially formulated RAM that gives a conflict free operation, which was not possible with simply generated RAM. Overall resource consumption of this block is less than 1% of overall available resources, and two block RAMs.

Now we will see the other signals which are changed and are asserted during RAM reading and word processing.

## 4.4 Processor-RAM block parallelism

In this research work we have attained higher speed of processing through multiple processors working in parallel. The processor-RAM block was instantiated many times within a module, that as whole was working as a single processor having multiple inputs and outputs. See the figure 4.4.

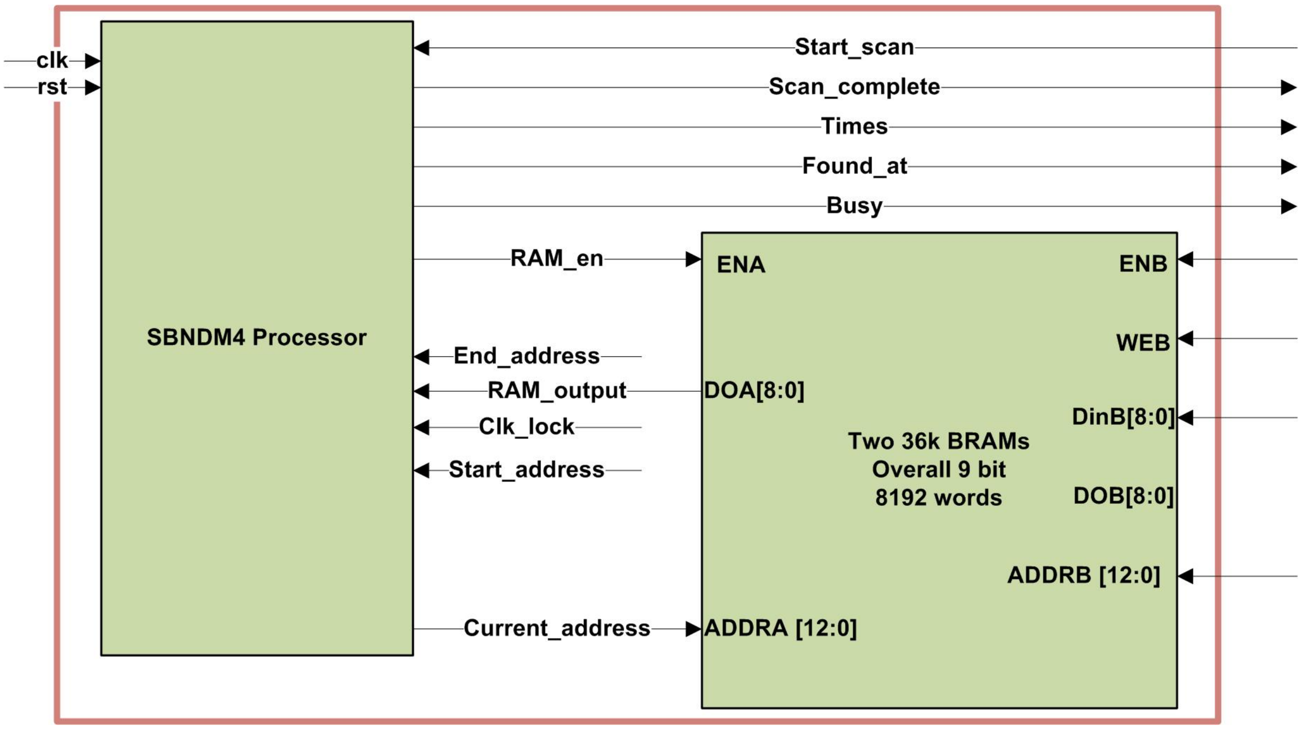


Figure 4.4: Processor-RAM block black box

In the figure above, we can see a box containing both a single processor and a single 72kb RAM. *Start_scan* is now connected as input to the block. *Scan_complete* and *Busy* signals are outputs of the block. Times and *Found_at* are now output from block.

We have instantiated the block in figure 4.4, 24 times in our design, and in result what we obtained is shown in figure 4.5. This is what we will refer to as replicated processors, and we will now discuss the end product of replication in detail. In figure 4.5, processor-RAM blocks are shown with names P0, P2,..,P23. The arrows in red color are common signals, which mean a single signal line will be given to all the replicated blocks. For example, Start_scan signal will be routed to all the blocks. This signal is stimulated by main controller. Next signal is *WEB*. This signal is same for all the blocks and is again controlled by main controller. *DinB* and *ADDRB* signals are also common, and are controlled by main controller.

### 4.4.1 Replication and signal adjustments

The arrows depicted in black color are all signals handled individually. *Times* is a signal reported by every processor individually. But, this is the result of a single processor. So, an adder is implemented so that when scan is completed, it also gives the accumulated effect of all the processor working in parallel. But, until the scan is not completed, adder results may vary, so to get final results, and to avoid any false reporting, adder will only trigger when scanning is completed by all the processors(see fig 4.5). *Scan_complete* is again handled individually. All individual scan complete signals depicts whether scan is completed by individual processor or not.

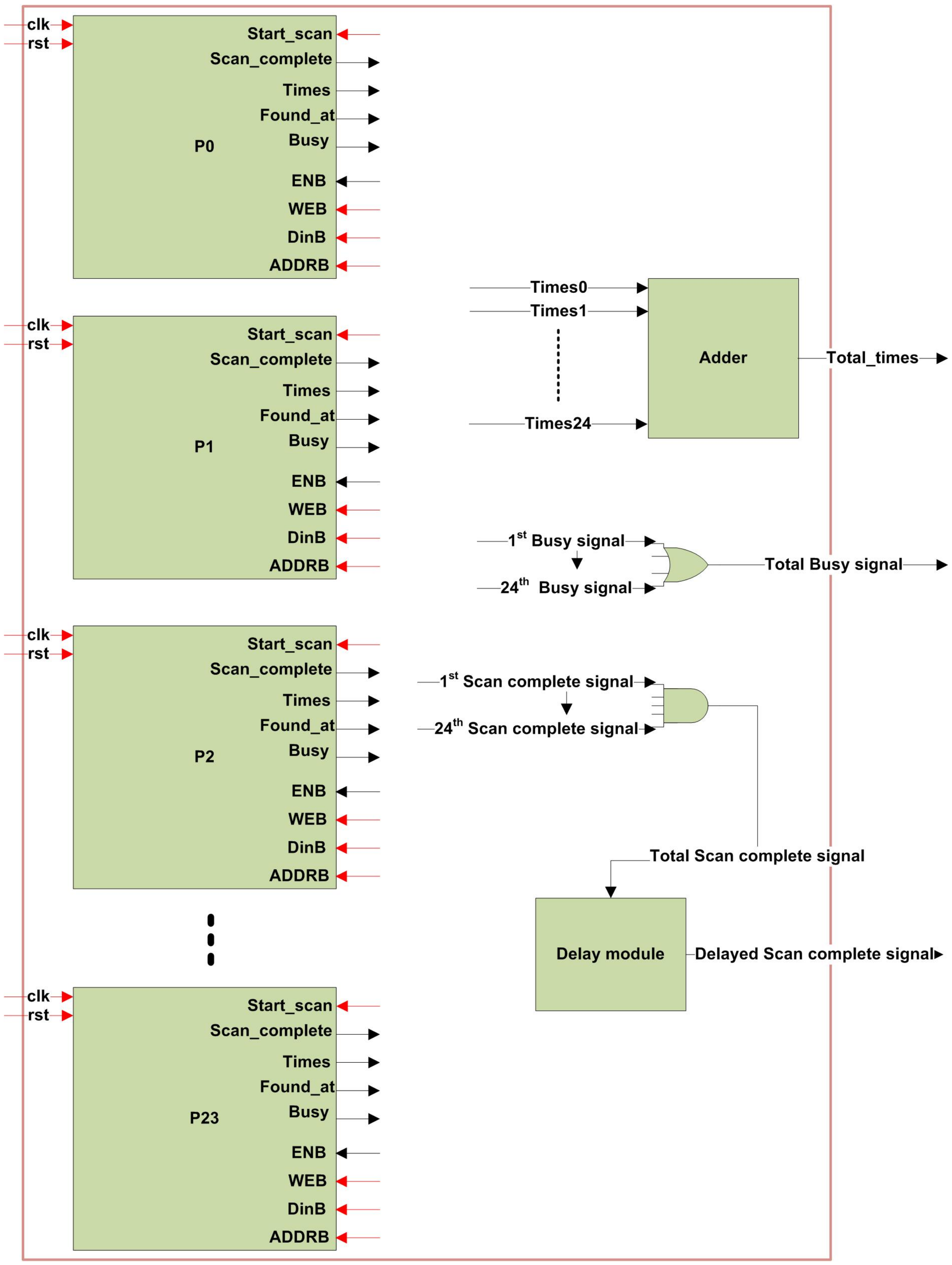


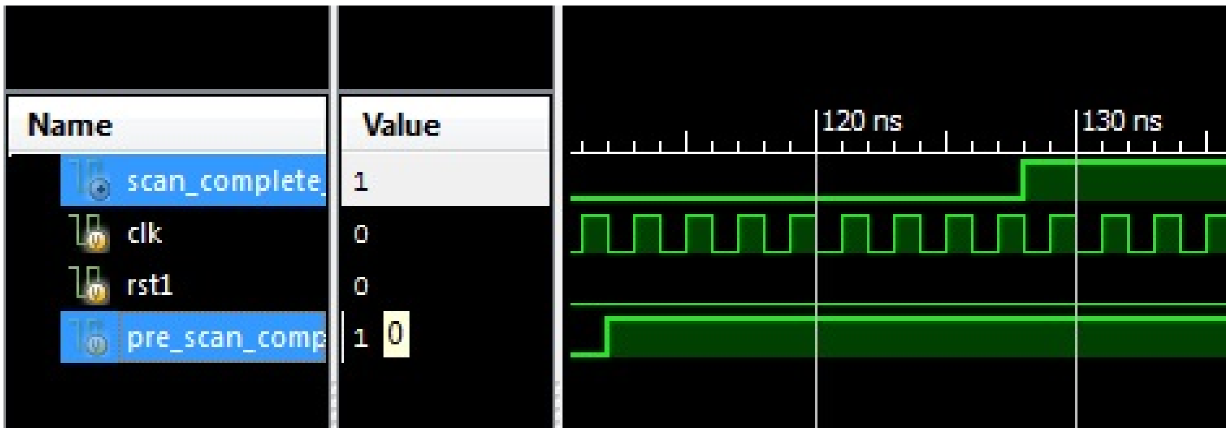
Figure 4.5: Processor-block replication and signals

But we are treating results as whole, so as long as any single processor is working, the results are not reliable, as results may be changed at any time. So all scan complete signals from individual processors are bounded by AND function so that, *total scan complete signal* is not getting high, if even a single processor is not finished scanning (see fig 4.5). Here, it may be noted that *total scan complete signal* is passing through a delay circuit. This delay circuit is implemented due to the fact that, output from individual processors is not read at once, rather cycle-by-cycle basis. So, whenever an output appears at processor's output, it takes some time to update that output to system output. But how much delay was optimum? By hits and trials, we came to a conclusion that for a worst case scenario, a delay of 8 system clock cycle was optimum, so that all outputs from all the processors are having a chance to get updated at systems output. This module is implemented with non-blocking assignments, and with a help of a separate Verilog module (see fig 4.5). *Busy_signal* shows that a particular processor is busy scanning a RAM. There are 24 processors working in parallel, so to combine the effects, we have used OR function to combine the effects of the processors. The output of this function will remain high, as far as even a single processor is busy. We have implemented this function with a simple Verilog primitive.

## 4.5 Testing and signal analysis

The design is tested with ISim, within Xilinx environment. Test benches are made to verify the functionality.

All RAMs are initialized with the help of .coe files. All RAMs are initialized using back-to-back pattern case, which is worst case, so that timing obtained are the best i.e. the system will always perform equal or better. We will first discuss delay circuit. Delay circuit is employed to give output updating circuitry a chance to read out all *Found_at* outputs to be read at system output RAM (will be discussed in next chapter). Delay circuit delays *scan_complete_signal* for 8 system clock cycles as can be seen in image below.



| Name | Value |
|---|---|
| scan_complete_ | 1 |
| clk | 0 |
| rst1 | 0 |
| pre_scan_comp | 1 0 |

Figure 4.6: Delay circuit output waveform demonstration

Now we will discuss about the results of processor replication. In processor replication the biggest issue is to align the signal and synchronize every operation taking place at every clock edge (positive or negative clock edge). As soon as the *Start_scan* signal is asserted, which is asserted after checking the status of one signal which is busy signal from multi-processor module, busy signal , that is accumulated effect of all the processors is asserted high. It may also be noted that the status of the *Locked_out* will also be checked in order to processors to start the scanning process. See figure 4.7.

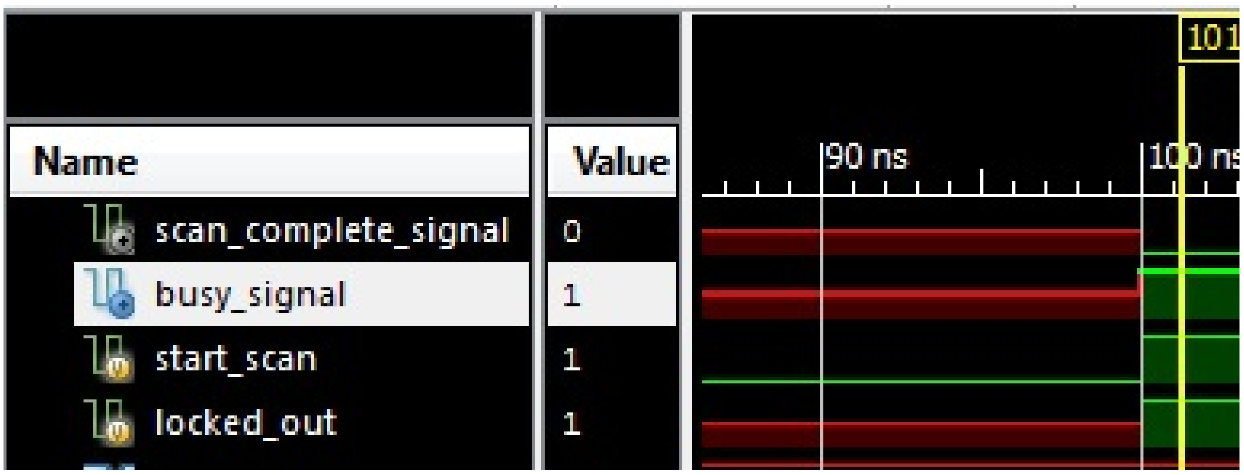


Figure 4.7: Start_scan signal on global scale

As soon as processors start to scan, results begin to show on *Found_at* and *Times* lines begin to update. Times circuit needs to updated at every clock cycle, so that when operation is completed , an updated value appears at *Times* output. This is depicted in figure 4.8.

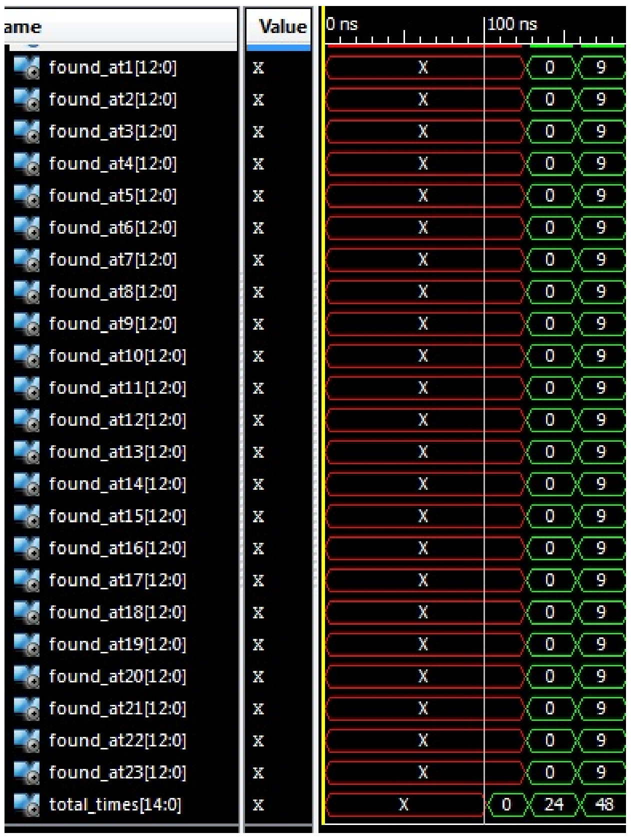


Figure 4.8: *Times* signal generation

In fig 4.8, it can be seen that every processor scanning a RAM is reporting that signal is found either at 0 or at address 9. But how will we know that where does that particular word lie's in actual text? We will discuss this matter in next chapter, where every RAM is assigned an special digit prefix, which at one side separates RAM addresses from other RAMs and also makes all RAMs to look like one big RAM.

After all the RAMs are scanned, during the very first positive half cycle, circuit tries to decide whether all processors are done scanning or not. As soon as all processors are done scanning, busy signal is asserted at very next negative clock edge. But the scan complete signal takes another 8 system clock cycles to get asserted positive. See fig 4.9 for reference. The effects of these 8 cycles will be discussed in detail in next chapter.

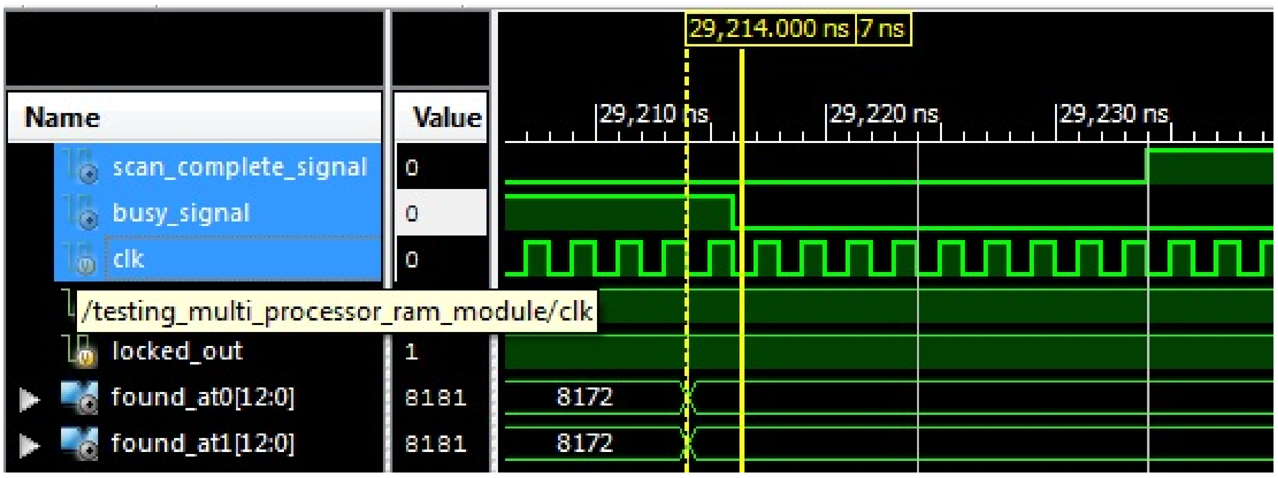


Figure 4.9: Scan complete signal

## 4.6 Summery and results

In this chapter, we saw the process of generation RAM cores through coregen tool within Xilinx ISE environment. We saw how to change the generated RAM core into a conflict free RAM, and into a shape that was required by our application. Then we made a processor-RAM block and brought the system to another higher level of abstraction. Then we demonstrated the concept of parallelism through instantiation of 24 SBNDM4 processor blocks. We saw how we connected different signals to get one accumulated system response. In last of the chapter we showed some waveform showing the sole idea of parallelism and resulting signal adjustments. Now we are at an abstraction level, where we will treat all instantiated processors as a single system identity, and we will continue to build system with this assumption from here on.

All the RAMs were scanned within around 29210ns, and scan complete signal asserted high around 29230ns. It was back-to-back case, that is one pattern followed by another with no chance of skipping characters(worst case possible). All the waveforms were generated keeping time scale at 10ns/1ps.

Now to compare the results we need to have the estimation of timing of writing all the RAMs , as our all BRAMs constitute a space that is less than 1M words. So in next chapter we will see how much time it takes to scanning all 1M words. Timings will include both scanning as well as writing RAMs, until it reaches 1M words limit, to compare our results from [1] to evaluate our work.

# Chapter 5

# Overall organization and results

*In this chapter we will see how different parts of our system are arranged. Some new parts and controllers will also be discussed and presented. Detailed analysis about the interfacing issues and their remedies will also be presented. In last of this chapter results of our research will be presented.*

In chapter 4, we saw how to replicate the processors to create the effects of parallelism. we also saw how to create signal on system basis, so that all processors are working as one entity.

As, all along the chapter 4 we saw the role of Main controller and output updating circuitry, we will see in detail now what these circuitry are.

## 5.1 Output controller

In the figure 5.1 an output updating module (also designed in Verilog) and a Output RAM is shown. These two parts are performing two distinctive operations. Output RAM Updating Circuit is coordinating with Processor block, from where it reads *Total_Times* outputs, and Output RAM holds the values, the address of patterns and total times it was found, and offers outside system with a port to read these value at any time after scanning is completed.

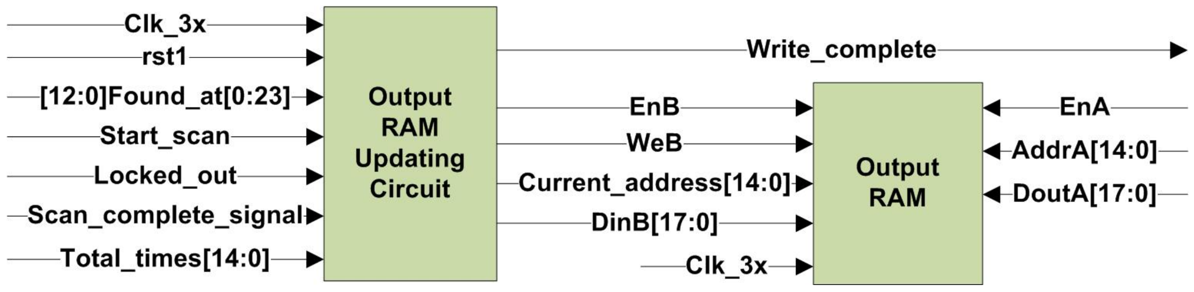


Figure 5.1: Output updating module and Output RAM

Output RAM updating or Output updating module is responsible for writing the *Found_at* and *Total_Time* outputs into output RAM. There are overall 24 *Found_at* outputs from 24 different processors, all canning 9 bit 8192 word RAMs. It is observed that shortest time any *Found_at* output is changing was almost 15 system clock cycles. But there were 24 such processors, and updating all in output RAM, so that we are not missing any findings, was not possible, when the output module was just running at system clock. So, by hit and trials it was detected that Output update module should run at the rate , 3 times the system clock is sufficient to enable output updating module to update RAM with every pattern found during scanning , by any processor, with 0% missing.

### 5.1.1 Output address update operation

Here we would like to mention one technique that we applied in order to make all 24 RAM to look like a single big RAM. As we know that every RAM was holding 8192 , 9 bit words. Address in all RAMs were from 0 to 8191. So, how to distinguish patterns found at different position in different RAMs? and how to tell where that pattern was found in overall text? The solution lies with-in the basic structure of truth tables. Every RAM was 8192 in depth, which mean 13 bits are more than sufficient to denote all address from 0 to 8191th address. There were 24 in total RAMs, so to distinguish these RAMs from each other we needed 5 more bits(11111b = 32d). So finally the Output update circuitry was reporting an address of width 18 bits. For example, we have found an address 0 of second RAM. So overall the address that will be reported will be $00001 - 0000000000000_b$ = $8192_d$. Similarly a pattern is found on $15^{th}$ address of fifth RAM, then the reported address will be $01111 - 0000000000100_b$ = $122884_d$ and so on. Here it may be noted that address $00000 - 0000000000000_b$ is reserved for Total_Times input, to be discussed in section 5.1.3.

### 5.1.2 Output RAM

As we saw in section 5.1.1, bit width of the content to be written in output RAM is 18 bits. There were 910 patterns in a single RAM, and overall there were 910 * 24 = 21840 patterns. So to design a RAM accommodating 18 bit, 21840 patterns(depth) we connected 11 BRAMs of 36kb space to form a single RAM. We generated a block RAM using Xilinx coregen tool. Then we connected this generated RAM in same manner as in fig 4.2. This time we will place B side of this RAM towards Output Update module, and will connect A to the terminal for outside system to read contents at any time after scanning is complete.

### 5.1.3 Output update signal sequence

Upon reception of *Start_scan* and Locked_out signals, Output controller, while working at 3X (section 5.2) clock rate than other system components, checks *Found_at* output from first[1] SBNDM4 processor. If this is same as before, then in next clock cycle it checks for second SBNDM4 processor (P1). This process continues till the last processor(P23) *Found_at* output. It may be noted that initially all Found_at outputs are assigned all ones value, so that if some processor detects a pattern at 0 or other RAM address, it is reported.

As soon as *Locked_out* and *Start_scan* signals are asserted, *WeB* and *EnB* signals along with the first address to the RAM to store addresses where the pattern was found are asserted to the output RAM. *Din* signal is asserted only when a pattern is found. When a second address is found, address to the RAM is updated with second place in output RAM as Din with the address where the pattern is found. The process goes on for the rest of the time. Along the process, Total_Times signal is updated at every clock cycle, an status of the Scan_complete signal is also checked. As soon as the Scan_complete signal is asserted high, controller places Total_Times at Din to be written at $0^{th}$ address in the output RAM (kept as default setting). After the RAM is written completely, *Write_complete* signal is asserted high as an indicator to outside system that contents in the output RAM are ready.

[1]Here first SBNDM4 processor is what marked as P0 in fig 4.5.

### 5.1.4 Testing of Output Updating Block

In the fig: 5.2, waveforms pertaining to Output updating module are shown. In the figure we have two signals *Start_scan* and *Locked_out* going high at once. At the very next negative clock edge, *current_address* is turned $0^{th}$ signal (corrected as $1^{st}$ address, while overall system integration, as $0^{th}$ place is reserved for *Total_times* signal ). In test bench, we have asserted all the *Found_at* signals at once, which the module updates on every successive negative clock edge. It must be noted that RAMs in our designs are working on the positive clock edges. That's the reason, why we have designed our system to work at negative clock edges.

After generating 24 addresses , at around 60ns, as soon as the *Scan_complete* signal goes high, on the very next edge, *WeB* and *EnB* signals are asserted low. On the very same time, logic levels on *Total_times* are presented at *DinB* output, and address is generated is also incremented(corrected as $0^{th}$ signal in final system integration).

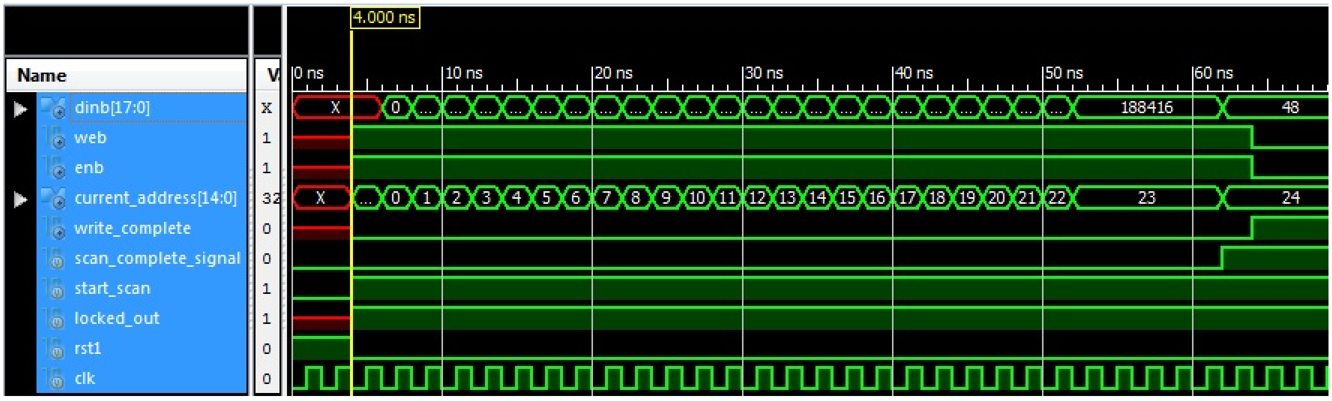


Figure 5.2: Output Updating module signals waveform

## 5.2 Xilinx DCM module

A Xilinx Digital clock manager block is shown in figure 5.3.

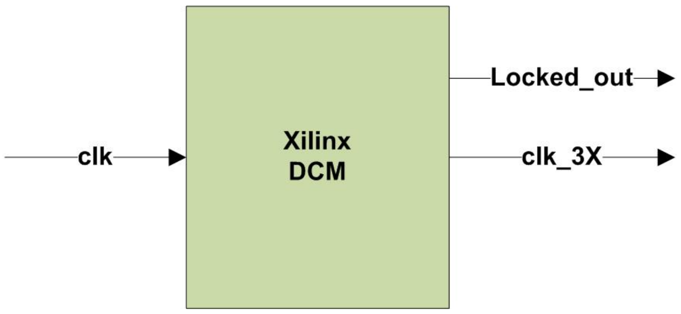


Figure 5.3: Xilinx DCM block

The DCM module takes system clock as input, and gives two outputs. One is a 3 times multiple of system clock, and other output is Locke_out, which is a sign of 3X output clock stability. This module was generated using Xilinx Coregen. *clkFx* was chosen so that we are having an option of multiplier and divisor to the input clock.

### 5.2.1 DCM testing and wave form

By 3 times clock we mean 3 cycles output per single system clock cycle. This may be seen in picture 5.4 . It may be noted that DCM takes some time to initialize, 3X clock output only appears or becomes stable after a brief delay. Furthermore, we will only use DCM output when *Locked_out* output is asserted high. All the modules of the system, which are starting their work with *Start_scan* signal, also checks for *Locked_out* signal, to start work at stable clock.

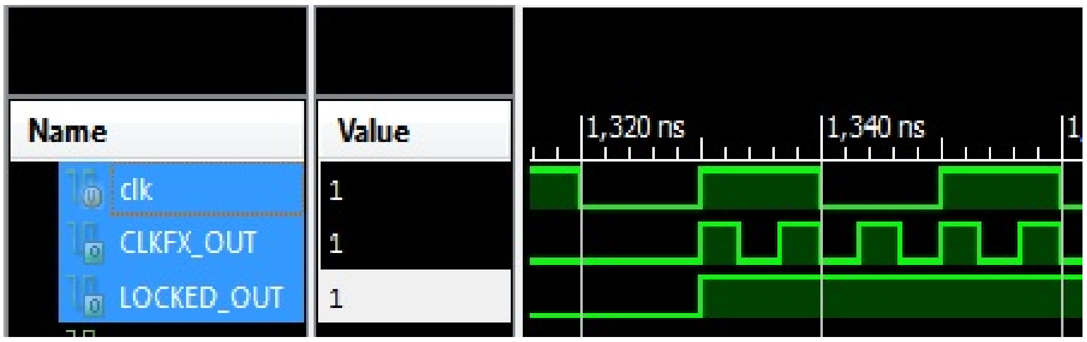


Figure 5.4: Xilinx DCM block 3X clock output

## 5.3 Main controller

Main controller is like a gateway of our system. It interacts with outside system and coordinates signal sequencing and synchronization between system inside and outside system. In our system, Main controller senses the signal from outside systems, and generates a system response for that. Similarly it also senses the signals generated from system inside and generates a response for that also. Block diagram of main controller is depicted in fig 5.5.

### 5.3.1 Main controller Signal sequencing

Main controller operates on normal system clock. One side of main controller is faces outside system. It is where it senses for *Data_ready* signal. This signal indicates that some data to be scanned is ready to be written within BRAM for scanning. When this signal is asserted high, next it senses for a signal facing insides system namely *busy_signal*. If this signal is low, main controller doesn't take next step. Once if , both the signals are asserted high and low respectively, main controller show's it willingness to do the job by asserting yet another signal namely *Reading*. If available data is assigned to any other system, outside system will not assert any of the signals. Main controller will wait for some time(8 clock cycles), and if *Ok_to_read* signal is asserted high, first address on *Address* line is asserted. Here we are assuming that data is also stored in some sort of memory somewhere outside system. On the very next clock cycle(negative edge) first address is asserted on *Current_address* line. This line goes from 0 to $8191^{th}$ and then back to 0. This is because EnB signal will be high for just one RAM at a time. As Address output is activated, meanwhile Main controller asserts WeB signal to all the RAMs. EnB signal is 25 bit wide, and it works on one-hot basis, i.e. only one bit will be high at a time and rest of the bits will be asserted low. So by that way, only one RAM can be written at any time. While updating Address line, main controller keeps a check whether address limit is reached upto address on *End_address* line. The data from outside world is 9 bit wide. DinB line is also 9 bit wide.

After the data read from outside world is complete, *Read_complete* signal is issued to the outside system to show that the read process is completed, and resources can now be relieved or be prepared for next usage.

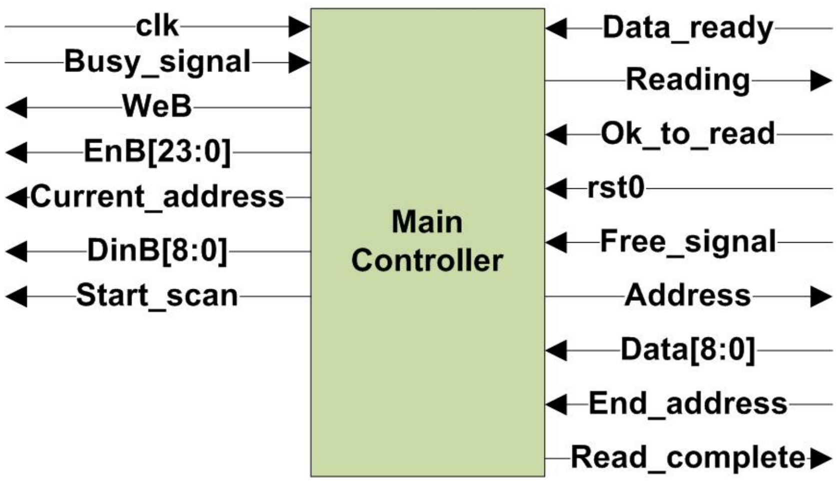


Figure 5.5: Main controller block diagram

At the same time *Free_signal* from outside system is also sensed, if this is asserted low, main controller will hold, till it is asserted high. As soon as *Free_signal* is asserted high, *Start_scan* is also asserted high. *Free_signal* shows that, the output RAM is free to be written, and all previous results have been retrieved from it.

It may be noted that we have devised here a MPI(message passing interface) between our system and outside system. This MPI can be replaced with any suitable protocol interface e.g. Ethernet, AXI or SPI etc.

## 5.3.2 Difference between rst1 and rst0

There are two kinds of resets given at the interface of the system. *rst0* is given to reset the logic in Processor sub-module and Main controller. *rst1* is given to reset Output updating module. The reason is that the two system although work together, but text scanning, and reading data from output RAM may be done separately. This option is given for more flexible system operation and utilizing resources at maximum.

## 5.3.3 Timing results and Waveforms

In the figure 5.6, it shows the series of signals that are produced when the operation of Main controller assumes to start. At the beginning as *data_ready* gets high, *reading* signal from controller also gets high. Here in the picture outside system is made to assert *Ok_to_read* signal after a brief moment of time, but in actual it may be asserted just after the *reading* signal gets high, or may get high after few cycles. As *Ok_to_read* signal is asserted high, controller asserts WeB high. Similarly EnB signal is also asserted 000000000000000000000001, making only first RAM enabled. Addresses on both sides of controller are updated with one cycle difference so that the word read from outside system are available on inside port to get stored in RAMs. It may be noted that *free_signal* was generated way before *data_ready* signal. *Start_scan* signal is still in asserted low, as RAMs are yet be written. *Read_complete* signal is also asserted low

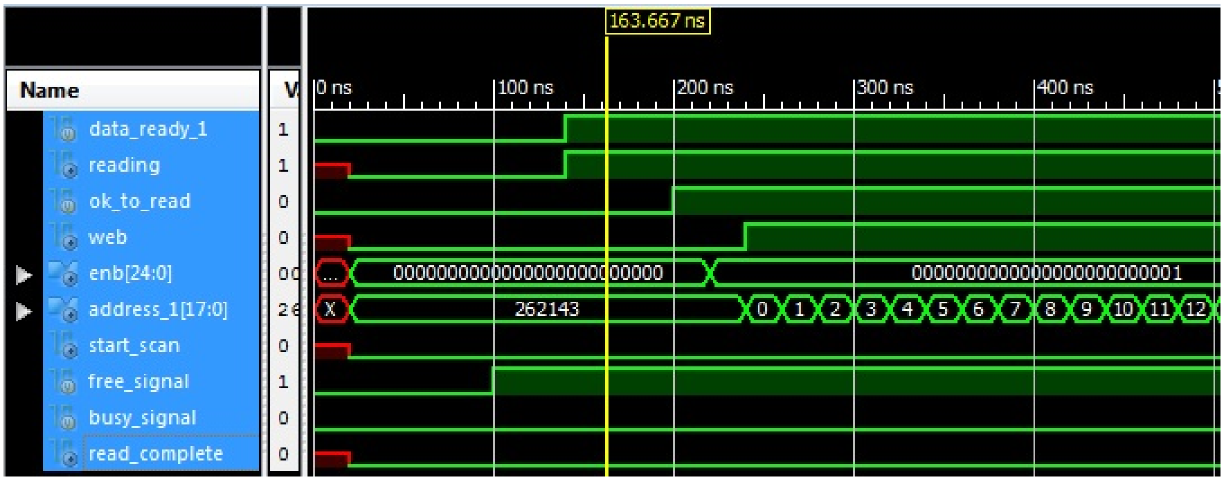


Figure 5.6: Main controller initial waveforms

to show outside system that our system is in process of reading words from outside system memory.

In figure 5.7 the final stage of operation is depicted. After all address are generated and all RAMs are written, WeB signal asserted low. At this point, EnB signal reaches the value 1000000000000000000000000, which mean non of the RAM is enabled for writing, as this 1 in pattern is at 25 place, which represents non of the RAMs. As the operation is completed, *Start_scan* signal is asserted high so that processors may start scanning operation, and output updating circuitry may start updating output RAM.

It took around 3.9329 ms to write in all RAMs. Final results are summarized in table 5.1.

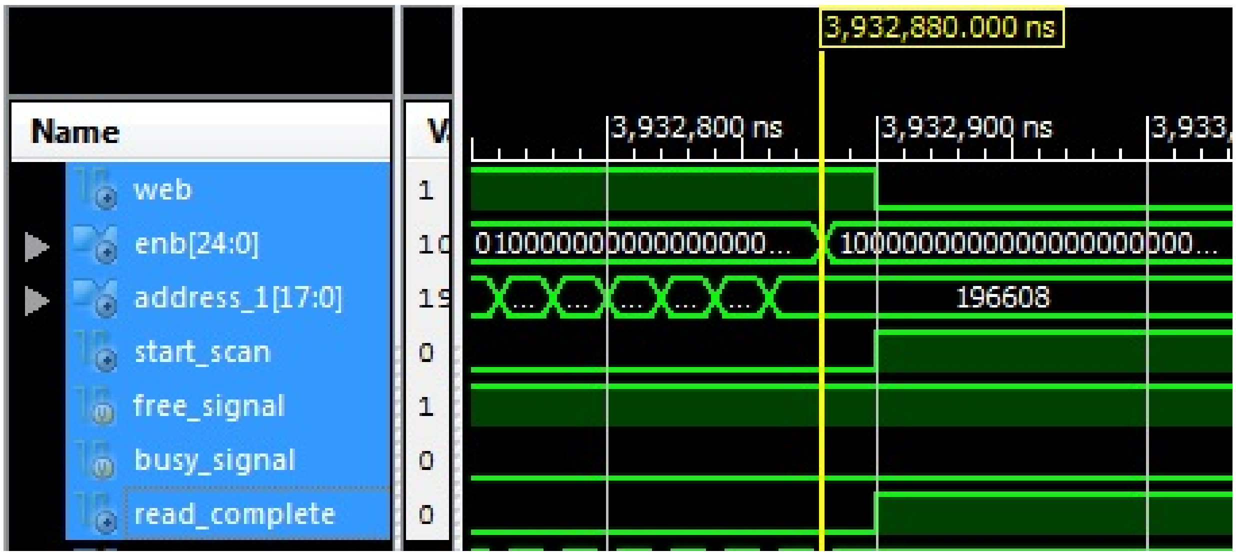


Figure 5.7: Main controller final waveforms

## 5.4 Integrating all system parts

In this part we will sum up all the parts together. All system parts integrated together are shown in figure 5.8. We start from Main controller that is connected to two domains, namely outside system and inside system. It senses data availability from outside system, when it senses data availability, it check whether inside system is busy, if not, then it requests outside system and by doing this renders its services to scan data. Outside system either responds or not against this request. If granted permission, Main controller begins to fill RAMs in Multi-processor module, which is further composed of 24 SBNDM4 processors and BRAMs. Main controller works on normal system clock.

The reason why there are 24 Processor modules is that we used BRAMs as buffer for text coming from outside world. There are limited number of RAMs available in FPGA chip which is 60. So, keeping a balance between numbers of RAMs per processor led us to reach a figure of 24, no even 25!.

As the RAMs get filled, Main controller issue a command to processor module to start scan. As the patterns are detected they get updated at found at outputs and here comes Output RAM Updating Module into play, which works at three times faster clock than normal system clock. The reason behind working at faster clock rate is that there are 24 processors, and all are producing outputs. So, if we will operate Output updating module at system clock rate then it is very likely that some of the detected patterns will not be updated in output RAM. For generating faster clock frequency, we utilized Xilinx DCM, which produces Lock signal along with faster clock frequency.

When all RAMs are scanned, and results are updated in output RAM, a signal to the outside system is generated that results are ready. At this point outside system issues rst0 command, and Main controller and Processor module assumes reset state. At this point Processor-RAM block RAMs may be re-written, but the main controller will not issue start scan command till it sense free signal to be asserted high. Free signal indicates that outside system is finished ready output RAM, and output RAM is free to be re-written. Outside system will issue rst1 command to Output updating module after reading output RAM, so that it also assumes it reset state.

At any given time our system can accommodate 192k words of 9 bit width. The range of addresses(start and end) are hard coded within Processor module. Only those values of addresses are given to modules which cannot be hard coded.

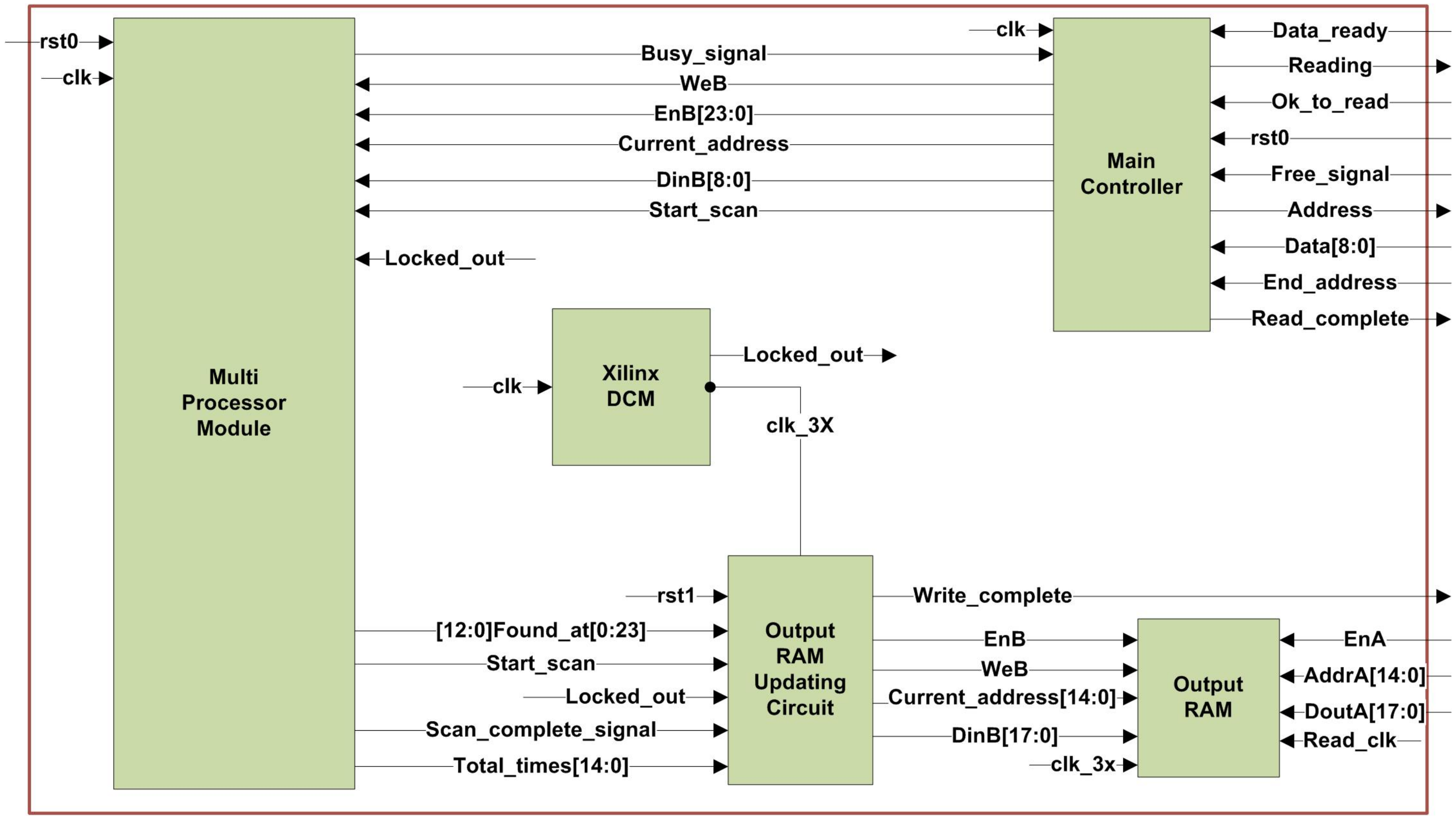


Figure 5.8: Overall system parts integrated

## 5.5 Results and comparison

Here are two types of results. In section 5.5.1 we will see how our timing results are better than the [1]. In section 5.5.2 we will see how much resources on FPGA are utilized, and what resources are left for further expansion.

### 5.5.1 Timing and acceleration

In [1], authors experimented with English text, of 10 character length , where data was in main memory, and the processors were working on 2.8 GHz. The processor was dual core. It took around 193ms to process 1M bytes or words.

In our case, we used XC5VLX50T-FFG1136F FPGA as processor, the data was retrieved through MPI, in chunks of 192k words, each word is 9 bit wide, and a target pattern of upto 9 characters. It took around 29.1714ms to process 1152k of such words. A total speedup of 6.6 times is achieved. Timing results are further elaborated in table 5.1.

| **Time elapsed**(ms) | **Time** (ms) | **Write** | **Scan** | **Over-all words Scanned** |
|---|---|---|---|---|
| | 3.9329 | ✓ | - | 192K |
| 4.8619 | 0.929 | - | ✓ | |
| 8.7948 | 3.9329 | ✓ | - | 384K |
| 9.7238 | 0.929 | - | ✓ | |
| 13.6567 | 3.9329 | ✓ | - | 576K |
| 14.5857 | 0.929 | - | ✓ | |
| 18.5186 | 3.9329 | ✓ | - | 768K |
| 19.4476 | 0.929 | - | ✓ | |
| 23.3805 | 3.9329 | ✓ | - | 960K |
| 24.3095 | 0.929 | - | ✓ | |
| 28.2424 | 3.9329 | ✓ | - | 1152K |
| 29.1714 | 0.929 | - | ✓ | |

Table 5.1: Final Results

### 5.5.2 Resource Utilization and ISE implementation

In the table 5.2, design utilization of FPGA is shown. It may be noted that slices are only utilized up to 7%, that means there is plenty of room for the accommodation of additional system built to expand the functionality of this system. DCM are utilized only 8%, and there are 11 more for other applications, specially if multi-rate systems are designed. Almost all Block RAMs are utilized, and if further RAMs are needed, we will make it in distributed logic.

The maximum design speed currently is 50MHz. This design speed may be increased if Only half block RAMs are utilized, as utilization of almost all the BRAMs is spreading design across all FPGA area, that is denting maximum frequency of operation. Similarly further optimization techniques may be applied to reduce Slice LUTs and Registers usage.

| **Slice logic Utilization** | **Used** | **Available** | **Utilization** |
|---|---|---|---|
| Number of Slice registers | 2282 | 28800 | 7% |
| Number of Slice LUTs | 7313 | 28800 | 25% |
| Number of Block RAM | 59 | 60 | 98% |
| Number of DCM_ADV | 1 | 12 | 8% |

Table 5.2: Final FPGA resource utilization

## 5.6 Summery

In this chapter we started to see things from the idea of macro system parts. We begin to see system parts from Output updating module, which is designed in Verilog. We saw how this circuit works in coordination with Processor module, which again was studied at macro level or higher level of abstraction. We saw how the operation is triggered by internal and external signals. We saw how External signals are connected to output RAM and we saw the sequence of reset proposed for this system. We saw why this module requires a higher clock frequency and how we generated it. Then we saw the usage of DCM with in Xilinx ISE, generated and used a DCM core. Then we saw how Main controller works, how it coordinates with outside system and internal system. How internal BRAMs are written, how scanning is done, how data is written in output RAM, and how it is indicated to outside system that results are ready, as well as the internal system is also ready to accept new text for performing another scan. We summarized our system in fig 5.8. In last of this chapter, we saw the timing performance of the system, and compared it with the [1], in which not only this algorithm was proposed, but also some experiment results were published. We saw that our system, while operating at lower frequency and utilizing very small space on FPGA, outperforms the results published in [1] by a factor of 6.6 .

# Chapter 6

# Future work and Conclusion

We concluded our work on a HDL instance , that is an implementation of SBNDM4 algorithm for scanning short patterns. We experimented with English text only. The results are promising, as well as easily implementable on other such platforms. In this thesis the main goal was to reduce that design size on chip so that remaining part of the chip may be used to other integral parts for the future expansions of the system. We used pure Programming language rather than system generator, to enable readability and for future use by other researchers. Our intention is to publish the work on open-forums like Opercores or Github etc. Next steps in the development of this project may be as one of the following:

## 6.1 As Computer-Accelerator pair

Our work may be used in a configuration depicted in figure 6.1.

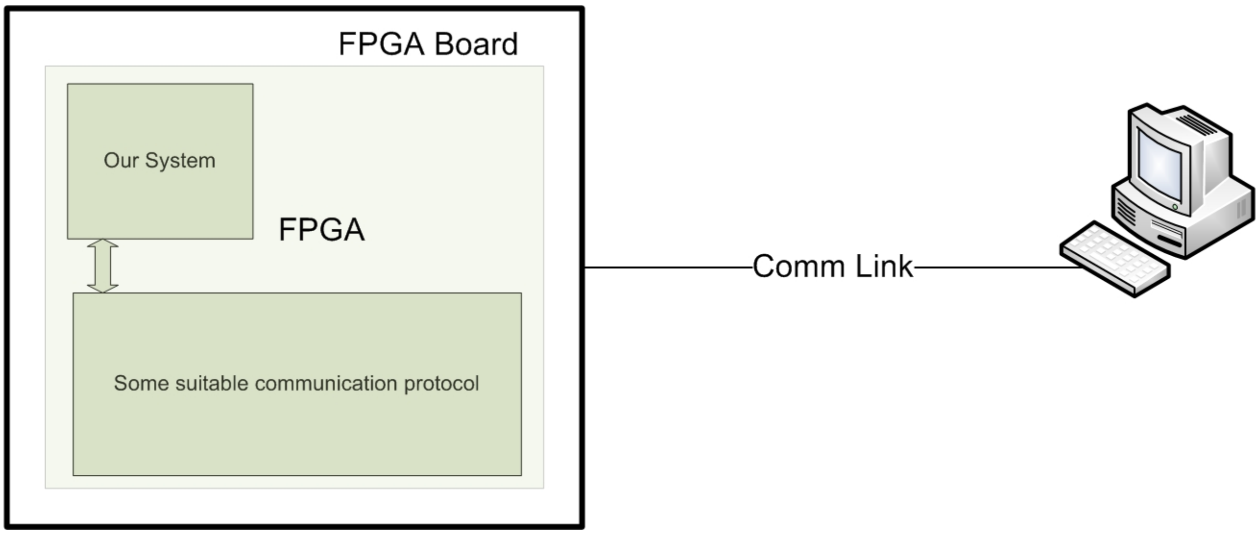


Figure 6.1: Proposed configuration 1

As in [57], our system may be downloaded in on an FPGA, after interfacing with suitable communication protocol HDL implementation(instance), this system may be used in query system or some short pattern detection from a given text. The suggested links for communication between FPGA board and PC are Gigabit Ethernet and PCI.

## 6.2 Data-base search

As in [50, 59, 66], out implementation may also be expanded as in figure 6.2. The SB-NDM4 system residing in a FPGA chip on a board, may be connected with a PC using suitable protocols, such a PCI or Ethernet. On the other end, the board may also be connected with a mass storage device, that may contain some sort of data base. The connection may be SATA or SCSI type, depending on the availability of ports on the board. Now, whenever the PC generates a query, the target pattern may be searched with in the data base and reported back.

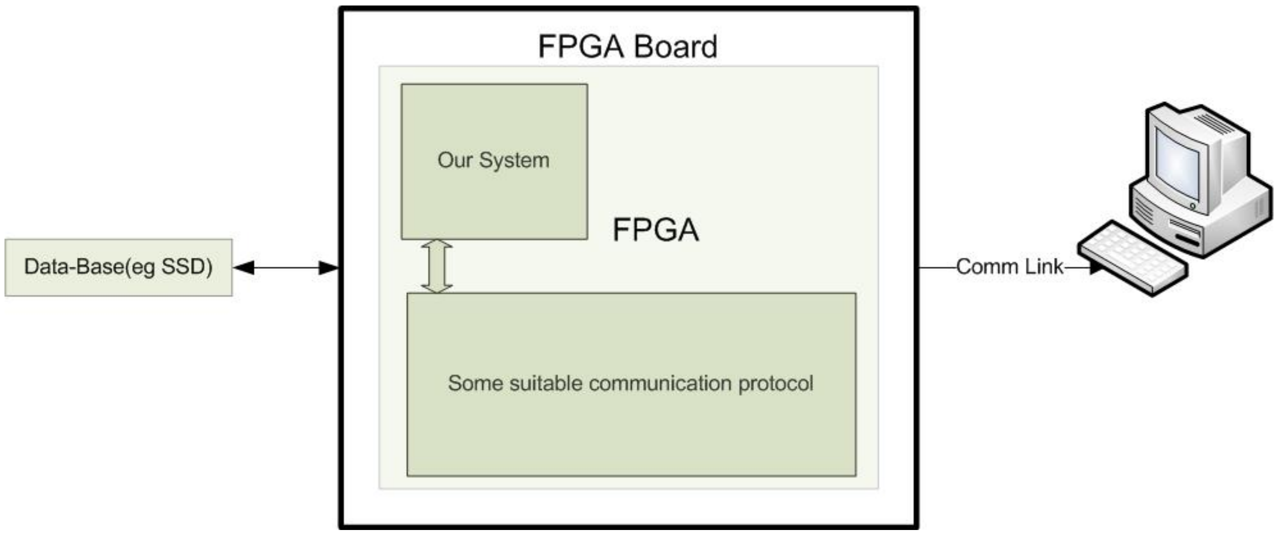


Figure 6.2: Proposed configuration 2

For some other application, where query generation is with quite high frequency, the system depicted in fig 6.3 may be adopted. In the configuration, another stem module with the name Run Time Manager is added, which is basically for the purpose of query management. This very same module may also be implemented on FPGA. A Run Time Manager not only will manage the query flow, but also the result flow towards computer.

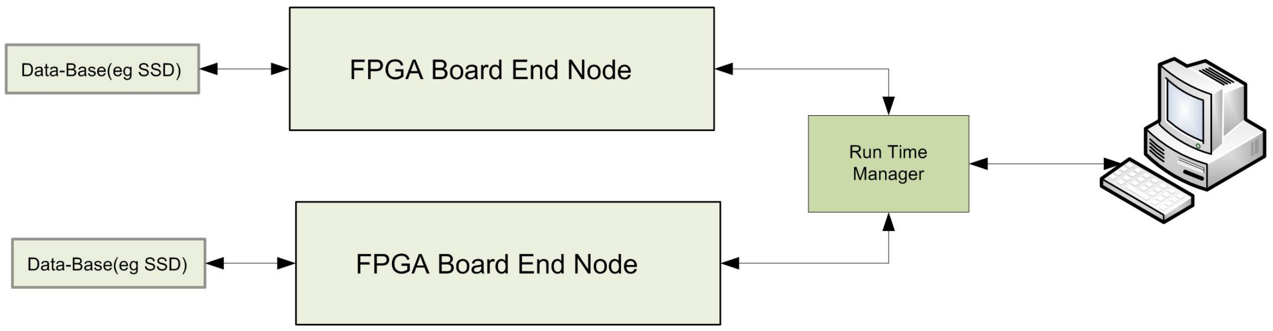


Figure 6.3: Proposed configuration 3

But this approach is not scalable as there are limited number of ports are available on a board. For this purpose another approach as depicted in figure 6.4. This image depicts a scalable structure. In this configuration, Run time Manager and SBNDM4 scanning system may be integrated into one FPGA. The FPGA board may require two Ethernet ports and one

SATA or SCSI or any other port to interface any mass storage device. A query generated by PC will be passed to the first such system. If that query is entertained by that system, then, second query will be sent. This second query is entertained by first system. If not, then it will be passed onto second system in line, and so on. It may be noted that each system in chain has its own Run time Manager, so, each system will responds to queries, will scan and route the results towards PC.

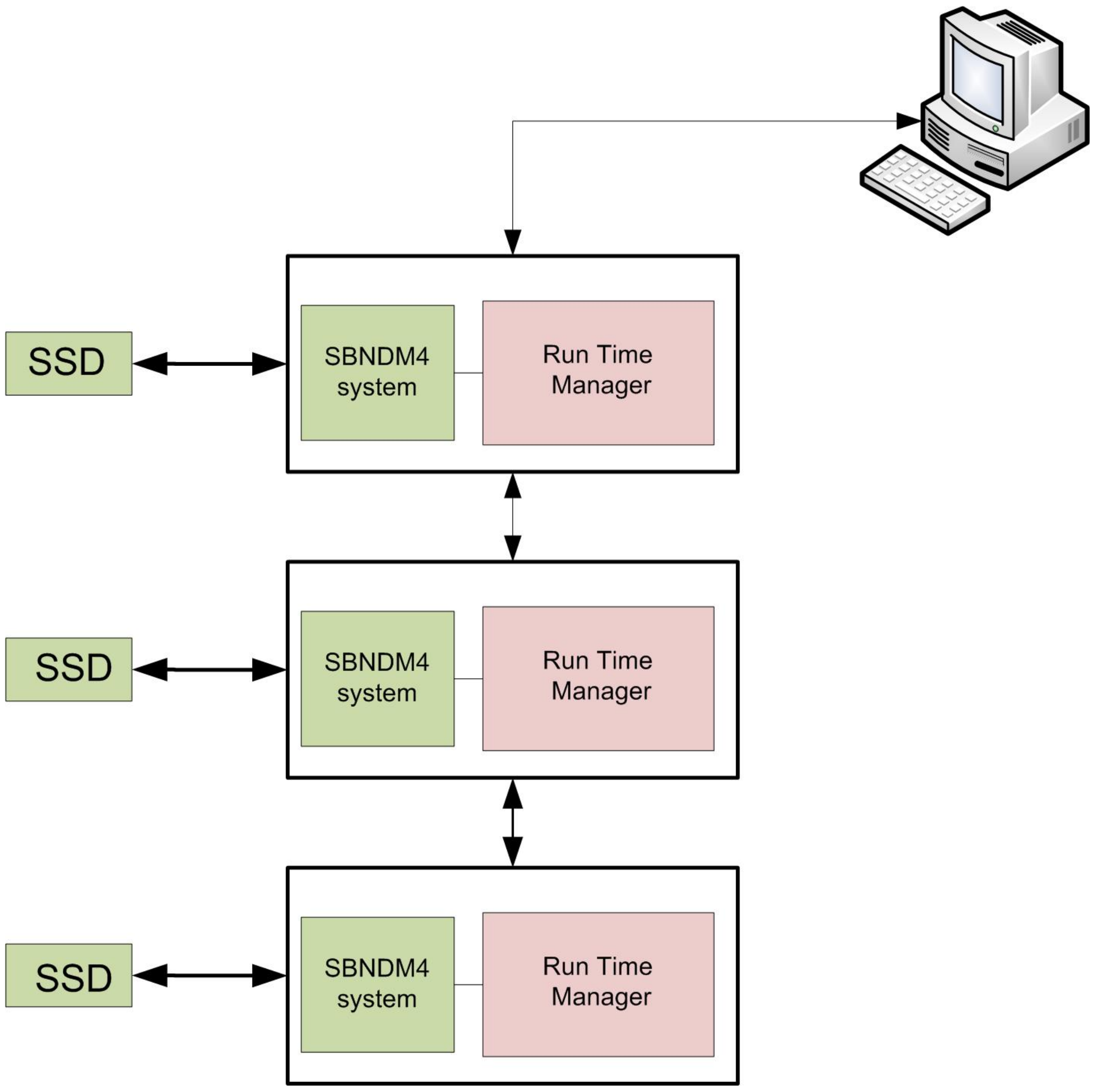


Figure 6.4: Proposed configuration 4

The down side of this approach will the speed limit imposed by the connection between first system in chain and the PC. As chain gets larger the traffic at this interface gets higher. So, there must be some optimize number of system to interface speed.

# References

[1] Branislav Durian, Jan Holub, Hannu Peltola and Jarma Tarhio, “Tuning BNDM with q-grams", In the proc. of workshop on algorithm engineering and experiments, SIAM, USA, pp. 29-37, 2009.

[2] J. S. Vetter, editor. “Contemporary High Performance Computing: From Petascale Toward Exascale", volume 1 of CRC Computational Science Series. Taylor and Francis, Boca Raton, 1 edition, 2013.

[3] What is high performance computing?, Retrieved from: http://insidehpc.com/hpc-basic-training/what-is-hpc/.

[4] Fide, S. and Jenks S. A Survey of String Matching Approaches in Hardware, University of California, Irvine, USA, Technical Report TR SPDS 06-01 , 2008, pp1-9.

[5] A. V. Aho and M. J. Corasick. “Efficient String Matching: An Aid to Bibliographic Search". Communications of the ACM, 18(6):333-340, 1975.

[6] Dandass YS, Burgess SC, Lawrence M, Bridges SM (2008) “Accelerating string set matching in FPGA hardware for bioinformatics research". BMC Bioinformatics 9: 197.

[7] Faro S. and Lecroq T, “ The exact online string matching problem: A review of the most recent results" , ACM Comput. Survey, Article 13, 42 pages, February 2013.

[8] D. E. Knuth, J. H. Morris, and V. R. Pratt. “Fast pattern matching in strings". SIAM Journal on Computing, 6(1):323-350, 1977.

[9] R. S. Boyer and J. S. Moore. “A fast string searching algorithm". Communications of the A CM, 20(10):762-772, 1977.

[10] G. Navarro and M. Raffinot. “A bit-parallel approach to suffix automata: fast extended string matching". In M. Farach-Colton, editor, Proc. of Combinatorial Pattern Matching, number 1448 in LNCS, pages 14-33, Rutgers, USA, 1998. Springer-Verlag.

[11] LogiCORE IP Block, Memory Generator. DS512. V7.1. Xilinx, April 24, 2012.

[12] R. Baeza-Yates, G.H. Gonnet, “A new approach to text searching", Comm. ACM 35 (10) (1992) 74-82.

[13] S. Wu, U. Manber, “Fast text searching allowing errors", Comm. ACM 35 (10) (1992) 83-91.

[14] G. Navarro and M. Raffinot, “Fast and flexible string matching by combining bit-parallelism and suffix automata", ACM Journal. Experimental Algorithmics 1998.

[15] Hannu Peltola and Jorma Tarhio, “ Alternative Algorithms for BitParallel String Matching", String Processing and Information Retrieval, Spire Springer, LNCS 2857, pp. 80-93, 2003.

[16] A. Blumer, J. Blumer, A. Ehrenfeucht, D. Haussler and R. McConnell, “Linear size finite automata for the set of all subwords of a word: An outline of results", Bull Europ. Assoc. Theoret Comput. Sci. 21 (1983) 12-20.

[17] M.E. Majster and A. Reiser, “Efficient on-line construction and correction of position trees", SIAM J. Comput. 9 (4) (1980) 785-807.

[18] D.R. Morrison, PATRICIA–“Practical algorithm to retrieve information coded in alphanumeric", J.ACM 15 (4) (1968) 514-534.

[19] V.R. Pratt, “Improvements and applications for the Weiner repetition finder", Unpublished manuscript, 1975.

[20] A.O. Slisenko, “String matching in real time: Some properties of the data structure", Prim 7th Syrup. on Mathematical Foundations of Computer Science, Lecture Notes in Computer Science 64 (Springer, Berlin, 1978) 493-496.

[21] A.O. Slisenko, “Determination in real time of all the periodicities in a word", Sov. Math. Dokl. 21 (2) (1980) 392-395.

[22] P. Weiner, “Linear pattern matching algorithms", IEEE 14th Ann. Syrup. on Switching and Automata Theory (1973) 1-11.

[23] A. Apostolico, “Some linear time algorithms for string statistics problems", Publication Series III, 176(IAC, Rome, 1979).

[24] A. Apostolico, “Fast applications of suffix trees", in: D.G. Lainiotis and N.S. Tzannes, eds., Advances in Control (Reidel, Hingham, MA, 1980) 558-567.

[25] A. Apostolico and F.P. Preparata, Optimal off-line detection of repetitions in a string, Theoret. Comput. Sci 22 (1983) 297-315.

[26] A. Apostolico, “The myriad virtues of suffix trees", Proc NATO Advanced Research Workshop on Combinatorial Algorithms on Words, Maratea, Italy, 1984.

[27] M. Rodeh, V.R. Pratt and S. Even, “Linear algorithm for data compression via string matching", J. ACM 28 (1) (1981) 16-24.

[28] S.L. Tanimoto, “A method for detecting structure in polygons", Pattern Recognition 13(6)(1981)389-394.

[29] R. Baeza-Y ates. “Text retrieval: Theory and practice". In 12th IFIP World Computer Congress, volume I, pages 465-476. Elsevier Science, September 1992.

[30] G. Navarro and M. Raffinot. “Fast and flexible string matching by combining bit-parallelism and suffix automata". ACM Journal of Experimental Algorithmics (JEA), 5(4), 2000.

[31] Faro, Simone, and Thierry Lecroq.“The exact string matching problem: a comprehensive experimental evaluation." arXiv preprint arXiv:1012.2547 (2010).

[32] R. A. Baeza-Yates and M. Regnier. “Average running time of the BoyerMoore-Horspool algorithm". Theor. Comput. Sci., 92(1):19-31, 1992.

[33] J. Holub and B. Durian. “Talk: Fast variants of bit parallel approach to suffix automata". In The Second Haifa Annual International Stringology Research Workshop of the Israeli Science Foundation, http://www.cri.haifa.ac.il/events/2005/string/presentations/Holub.pdf , 2005.

[34] L. He, B. Fang, and J. Sui. “The wide window string matching algorithm". Theor. Comput. Sci., 332(1-3):391-404, 2005.

[35] K. Fredriksson and S. Grabowski. “Practical and optimal string matching". In M. P. Consens and G. Navarro, editors, SPIRE, volume 3772 of Lecture Notes in Computer Science, pages 376-387. Springer-Verlag, Berlin, 2005.

[36] M. Oğuzhan Külekci. “A method to overcome computer word size limitation in bit-parallel pattern matching". In S.-H. Hong, H. Nagamochi, and T. Fukunaga, editors, Proceedings of the 19th International Symposium on Algorithms and Computation, ISAAC 2008, volume 5369 of Lecture Notes in Computer Science, pages 496-506, Gold Coast, Australia, 2008. Springer-Verlag, Berlin.

[37] S. Faro and T. Lecroq. Efficient variants of the Backward-Oracle-Matching algorithm. In Jan Holub and Jan Žďárek, editors, Proceedings of the Prague Stringology Conference 2008, pages 146-160, Czech Technical University in Prague, Czech Republic, 2008.

[38] G. Zhang, E. Zhu, L. Mao, and M. Yin. “A bit-parallel exact string matching algorithm for small alphabet". In X. Deng, J. E. Hopcroft, and J. Xue, editors, Proceedings of the Third International Workshop on Frontiers in Algorithmics, FAW 2009, Hefei, China, volume 5598 of Lecture Notes in Computer Science, pages 336-345. Springer-Verlag, Berlin, 2009.

[39] D. Cantone, S. Faro, and E. Giaquinta. Bit-(parallelism)$^2$: “Getting to the next level of parallelism". In Paolo Boldi and Luisa Gargano, editors, Fun with Algorithms, volume 6099 of Lecture Notes in Computer Science, pages 166-177. Springer-Verlag, Berlin, 2010.

[40] D. Cantone, S. Faro, and E. Giaquinta. “A compact representation of nondeterministic (suffix) automata for the bit-parallel approach". In A. Amir and L. Parida, editors, Combinatorial Pattern Matching, volume 6129 of Lecture Notes in Computer Science, pages 288-298. Springer-Verlag, Berlin, 2010.

[41] William Dally , Brian Towles, “Principles and Practices of Interconnection Networks". Morgan Kaufmann Publishers Inc., San Francisco, CA, 2003.

[42] Q. Zhang, R.D. Chamberlain, R.S. Indeck, B.M. West, J. White, Massively parallel data mining using reconfigurable hardware: Approximate string matching, in: Proceedings of Workshop on Massively Parallel Processing, 2004, p. 259a.

[43] M. Necker, D. Contis and D.E. Schimmel, “TCP-Stream reassembly and state tracking in hardware", Proceedings of IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM), April 2002, Napa, CA, USA, pp. 286-287, 2002.

[44] S. Li, J. Toressen and O. Soraasen, “Exploiting stateful inspection of network security in reconfigurable hardware", Proceedings of International Conference on Field Programmable Logic and Applications (FPL), September 2003, Lisbon, Portugal, pp. 1153-157, 2003.

[45] D. Schuehler and J. Lockwood, “A modular system for FPGA-based TCP flow processing in high-speed networks", Proceedings of International Conference on Field Programmable Logic and Applications (FPL), 00, pp. 301-310, Antwerp, Belgium, August 2004.

[46] R. Franklin, D. Carver and B.L. Hutchings, “Assisting network intrusion detection with reconfigurable hardware", Proceedings of IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM), April 2002, Napa, CA, USA, pp. 111-120, 2002.

[47] S. Dharmapurikar, P. Krishnamurthy, T. Sproull and J.W. Lockwood, “Deep packet inspection using parallel bloom filters", Proceedings of Symposium on High Performance Interconnects (HotI), Stanford, CA, USA, pp. 44-51, 2003.

[48] M. Gokhale, D. Dubois, A. Dubois, M. Boorman, S. Poole and V. Hogsett, “Granidt: Towards gigabit rate network intrusion detection technology", Proceedings of International Conference on Field Programmable Logic and Applications (FPL), September 2002, Montpellier, France, pp. 404-413, 2002.

[49] C.R. Clark, W. Lee, D.E. Schimmel, D. Contis, M. Koné and A. Thomas, “A hardware platform for network intrusion detection and prevention", Proceedings of Workshop on Network Processors and Applications at HPCA (NP-3), February 2004, Madrid, Spain, pp. 136-145, 2004.

[50] J.W. Lockwood, C. Neely, C. Zuver, J. Moscola, S. Dharmapurikar and D. Lim, “An extensible, system-on-programmable-chip, content-aware internet firewall", Proceedings of International Conference on Field Programmable Logic and Applications (FPL), April 2003, Napa, CA, USA, pp. 859-868, 2003.

[51] Lavenier, Dominique, and Van-Hoa Nguyen. "Seed-Based Parallel Protein Sequence Comparison Combining Multithreading, GPU, and FPGA Technologies." Bioinformatics: High Performance Parallel Computer Architectures (2010): 181-202.

[52] UniProt Consortium, The Universal Protein Resource (UniProt), release 2016_10.

[53] Shendure J, Hanlee J, “Next-generation DNA sequencing". Nature Biotechnology, 26(10): 1135-1145, 2008.

[54] Pearson W, Lipman D, “Improved tools for biological sequence comparison". Proceedings of the National Academy of Science, 85(8): 2444-2448, 1988.

[55] Altschul S, Gish W, Miller W, Myers E, Lipman D, “Basic local alignment search tool". Journal of Molecular Biology, 215(3): 403-410, 1990.

[56] Muriki K, Underwood K, Sass R, “RC-BLAST: towards a portable, cost-effective open source hardware implementation". In 19th International Parallel and Distributed Processing Symposium IPDPS05, 2005.

[57] Lancaster J, Jacob A, Buhler J, Harris B, Chamberlain R,“ Mercury BLASTP: accelerating protein sequence alignment". ACM Transactions on Reconfgurable Technology and Systems, 1(2), 2008.

[58] Lavenier D, Gille Georges G, Xinchu L, “ A reconfgurable index in flash memory tailored to seed-based genomic sequence comparison algorithms". VLSI Signal Processing. 48(3): 255-269, 2007.

[59] Server Kasap , Khaled Benkrid , Ying Liu, “A high performance fpga-based implementation of position specific iterated blast". Proceedings of the ACM/SIGDA international symposium on Field programmable gate arrays, February 22-24, 2009, Monterey, California, USA.

[60] C.-C. Chung, C.-K. Liu, and D.-H. Lee,“Fpga-based accelerator platform for big data matrix processing". Electron Devices and Solid-State Circuits (EDSSC), 2015 IEEE International Conference on, pp. 221-224, IEEE, 2015.

[61] de Souza, Viviane LS, Victor WC de Medeiros, and Manoel E. de Lima.“Architecture for dense matrix multiplication on a high-performance reconfigurable system". Proceedings of the 22nd Annual Symposium on Integrated Circuits and System Design: Chip on the Dunes. ACM, 2009.

[62] Holanda, B., et al. “An FPGA-based accelerator to speed-up matrix multiplication of floating point operations". Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on. IEEE, 2011.

[63] Dave, Nirav, et al.“Hardware acceleration of matrix multiplication on a xilinx fpga". 5th IEEE/ACM International Conference on Formal Methods and Models for Codesign (MEMOCODE 2007). IEEE, 2007.

[64] Bensaali, Faycal, Abbes Amira, and Reza Sotudeh. “Floating-point matrix product on FPGA". In: Procs of the IEEE/ACS Int Conf on Computer Systems and Applications (AICCSA’07). IEEE, 2007.

[65] Matam, Kiran Kumar, and Ming Hsieh. “Evaluating energy efficiency of floating point matrix multiplication on FPGAs". HPEC. 2013.

[66] Jie Zhang, Xiaoshan Jiang, Jie Wu, Jie Shan. “NetStorageFPGA-A Prototyping Platform for Building High-performance Transmission and Storage Systems Using Field Programmable Gate Array (FPGA) Hardware". Real Time Conference (RT), 2014 19th IEEE-NPSS. IEEE, 2014.

[67] Moorthy, Theepan, Sathish Gopalakrishnan. “Gigabyte-Scale Alignment Acceleration of Biological Sequences via Ethernet Streaming". International Conference on Field-Programmable Technology (FPT), 2014. IEEE, 2014.

# Appendix A

# Verilog Codes For Different Parts of The System

Please see the attached CD to find the codes for following parts:

- SBNDM4 Processor
- Main Controller
- Output Controller
- Generated cores for buffer RAMS
- Generated core for output RAM
- Generated core for DCM
- Complete Project in Xilinx ISE

Please also find the soft copy of this thesis in PDF form.

This page is intentionally left blank.