Chapter 5

Partial Encoding Architecture

Although the architecture we proposed seems effective, an enormous problem has been raised as the number of characters per symbol (i.e. q) grows. With the increasing number of characters per symbol, the alphabet size will rapidly expand. Hence, the symbol encoder occupies tons of logic elements as the alphabet size increase. Therefore, the proposed architecture is not suitable with the condition of increasing alphabet size.

An architecture called *Partial Encoding Architecture* [13] can exactly solve the problem of too much logic element consumption by the symbol encoder.

5.1 High Throughput Module Circuit with $q = 2$

As shown in Figure 5.1, a pattern "*abcde*" is given as the example. Consider the odd position and the even position, where all possible two-character combinations can be mapped from "*ab*","*bc*", "*cd*", "*de*" and others into x_4, x_3, x_2, x_1 and x_0 respectively by encoder f as illustrated in Figure 5.1(a). We call f, the full encoder. In addition, since the first and the second character are position dependent, we can consider mapping the individual character separately to obtain the partial encoding results. The two-character full encoder f can be decomposited into two single-character partial encoders called encoder1 f_1 and encoder2 f_2. As shown in Figure 5.1(b), the first character can be mapped to the encoder1 called $f_1()$. Similarly, the second character can be mapped to the encoder2 called $f_2()$

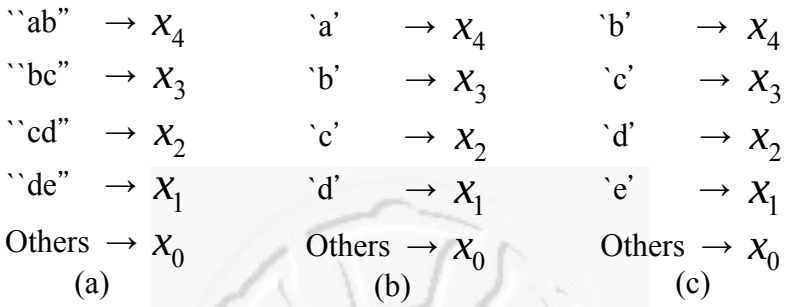

Figure 5.1: A pattern "*abcde*" can be decomposited from a two-character encoder into two single-character encoders, (a)Two-character encoder f (full encoder) (b)Single-character encoder1 f_1 (partial encoder) (c)Single-character partial encoder2 f_2 (partial encoder)

as illustrated in Figure 5.1(c). We can observe from the figure that the full result of "bc" mapped to symbol x_3 can be divided into $f_1(`b')$ and $f_2(`c')$.

Consider an input sequence $t_1 t_2 \ldots t_n t_{n+1} \ldots$ to the circuit. Our objective is to compute $f(t_{n-1}, t_n)$ and $f(t_n, t_{n+1})$ within the same clock cycle using partial encoder f_1 and f_2. One way to solve this problem is shown in Figure 5.2. Within the first clock cycle, we want to get $f(t_{n-1}, t_n)$ and $f(t_n, t_{n+1})$ in the rising and falling edge, respectively. Next, during the second clock cycle of rising and falling edge, $f(t_{n+1}, t_{n+2})$ and $f(t_{n+2}, t_{n+3})$ are also desired to be obtain.

Figure 5.3 shows the corresponding circuit. Within the circuit, only one copy of single-character encoder1 f_1 and two copies of single-character encoder2 f_2 are utilized. Actually, the dataflow of the partial encoding circuit is similar to DDR

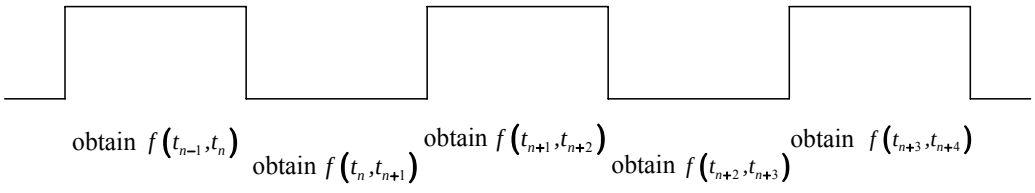

Figure 5.2: The timing diagram for the partial encoding architecture with $q = 2$

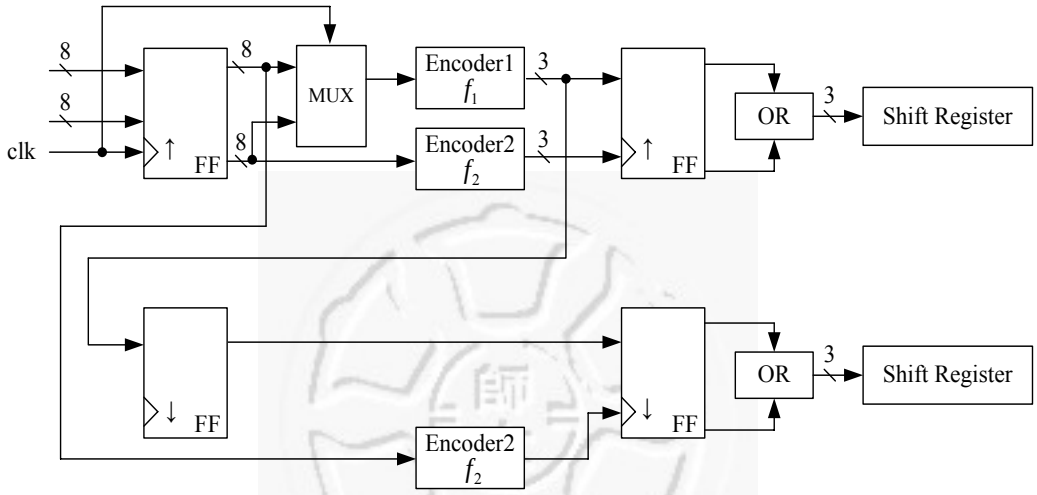

Figure 5.3: The partial encoding circuit with $q = 2$

RAM. A clock cycle is divided into rising edge and falling edge. As demostrated in Figure 5.4(a), during the rising edge of the first clock cycle, t_n is fed into $f_2()$ to get $f_2(t_n)$. t_{n+1} is delivered into $f_1()$ to get $f_1(t_{n+1})$. At this moment, flip-flop releases $f_1(t_{n-1})$. The full encoding function $f_(t_{n-1}, t_n)$ can be formed by *ORing* the partial encoder $f_1(t_{n-1})$ and $f_2(t_n)$. Then, the value of the full encoding function $f(t_{n-1}, t_n)$ will be delivered to the shift register for the bitmap encoding operation.

The falling edge of the first clock cycle is illustrated in Figure 5.4(b). t_n is fed into $f_1()$ to get $f_1(t_n)$. t_{n+1} is fed into $f_2()$ to get $f_2(t_{n+1})$. Then, $f_1(t_n)$ and $f_2(t_{n+1})$ forms the full encoding function $f(t_n, t_{n+1})$ by *OR* operation. Then, the value of the full encoding function will be sent to the shift register for the bitmap encoding operation.

As the characters per symbol grows, the partial encoding architecture can perform much more productive in saving the area cost of the symbol encoder. To elaborate on the concept, the extension from $q = 2$ to $q = 4$ will be discussed in the next section.

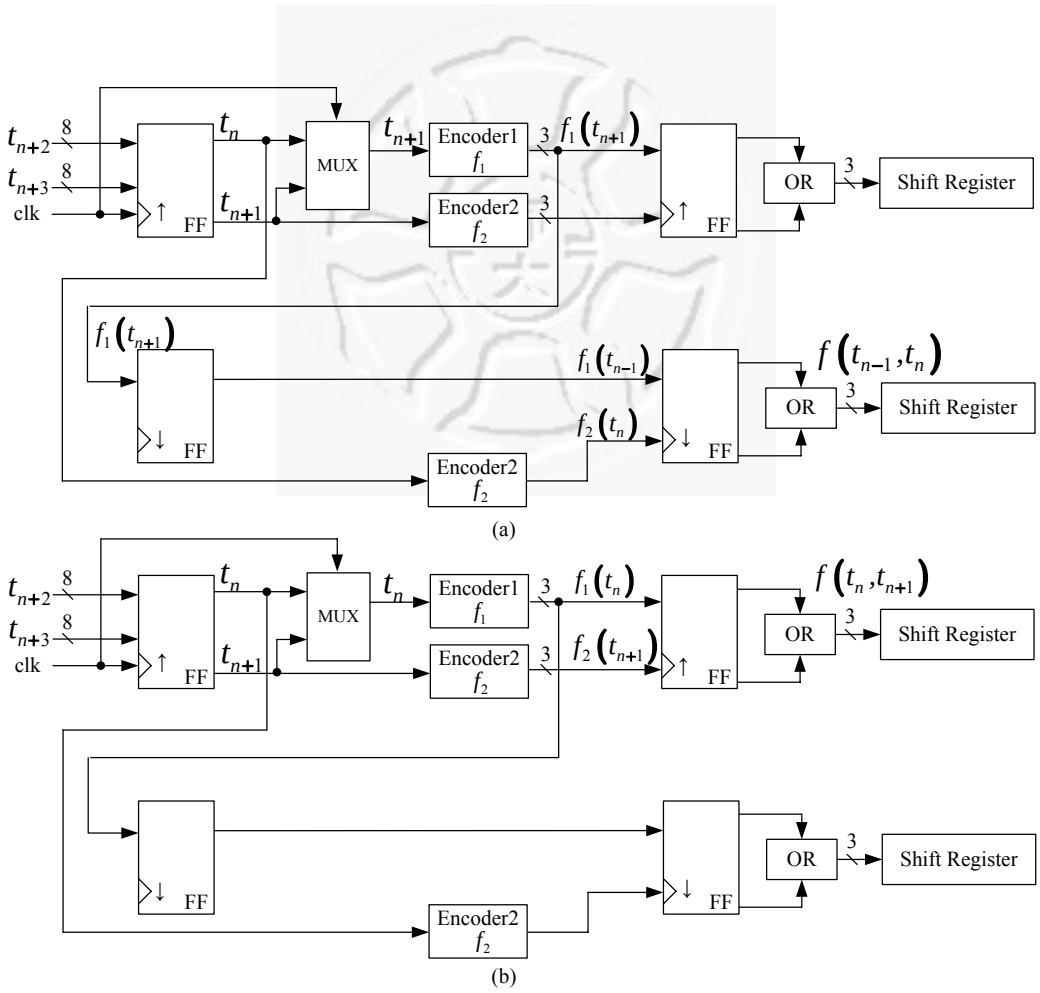

Figure 5.4: Dataflow of the partial encoding circuit with $q = 2$, (a) Rising edge (Obtain $f(t_{n-1}, t_n)$) (b) Falling edge (Obtain $f(t_n, t_{n+1})$)

5.2 High Throughput Module Circuit with $q = 4$

Figure 5.6 demostrates the timing diagram of our objective result with $q = 4$. Our objective is to compute $f(t_{n-3}, t_{n-2}, t_{n-1}, t_n)$, $f(t_{n-2}, t_{n-1}, t_n, t_{n+1})$, $f(t_{n-1}, t_n, t_{n+1}, t_{n+2})$, $f(t_n, t_{n+1}, t_{n+2}, t_{n+3})$ within the same clock cycle using partial encoders. One simple approach to solve the problem is shown in Figure 5.5. During the rising edge, we want to obtain $f(t_{n-3}, t_{n-2}, t_{n-1}, t_n)$ and $f(t_{n-1}, t_n, t_{n+1}, t_{n+2})$ in the circuit. Then, the value of $f(t_{n-2}, t_{n-1}, t_n, t_{n+1})$ and $f(t_n, t_{n+1}, t_{n+2}, t_{n+3})$ are desired in the falling edge.

In the same manner as $q = 2$, the four-character symbol encoder can be decomposited into four two-character symbol encoders. The circuit contains two encoder1 f_1 and two encoder2 f_2 as shown in Figure 5.6. The dataflow of the circuit is similar to the DDR RAM concept, a clock cycle is divided into rising edge and falling edge. As demostrated in Figure 5.7(a), during the rising edge, t_n, t_{n+1}, t_{n+2}, t_{n+3} are delivered into two $f_1()$, $f_2()$ to obtain $f_1(t_{n-1}, t_n)$, $f_2(t_{n+1}, t_{n+2})$, $f_1(t_{n-3}, t_{n-2})$, $f_2(t_{n-1}, t_n)$, respectively. In addition, $f_1(t_{n-1}, t_n)$, $f_2(t_{n+1}, t_{n+2})$ forms the full encoding function $f(t_{n-1}, t_n, t_{n+1}, t_{n+2})$ by OR operation in the rising edge. At this time, the full encoding function $f(t_{n-3}, t_{n-2}, t_{n-1}, t_n)$ is also constructed by $f_1(t_{n-3}, t_{n-2})$ and $f_2(t_{n-1}, t_n)$. Then, all the full encoding function will be delivered to the shift register for the bitmap encoding operation.

As illustrated in Figure 5.7(b), during the falling edge, the full encoding function $f_(t_n, t_{n+1}, t_{n+2}, t_{n+3})$ is constructed by the partial encoders $f_1(t_n, t_{n+1})$ and

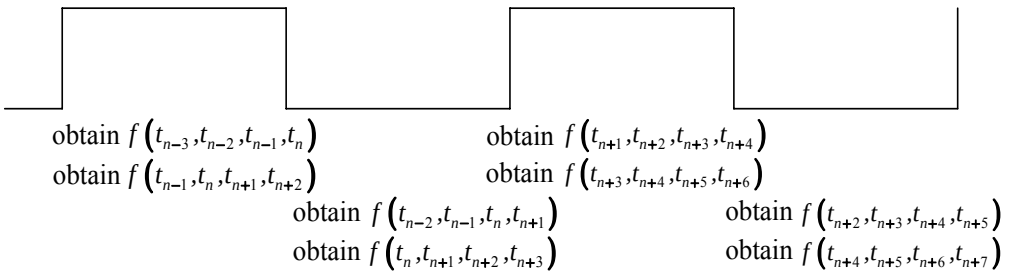

Figure 5.5: The timing diagram for the partial encoding architecture with $q = 4$

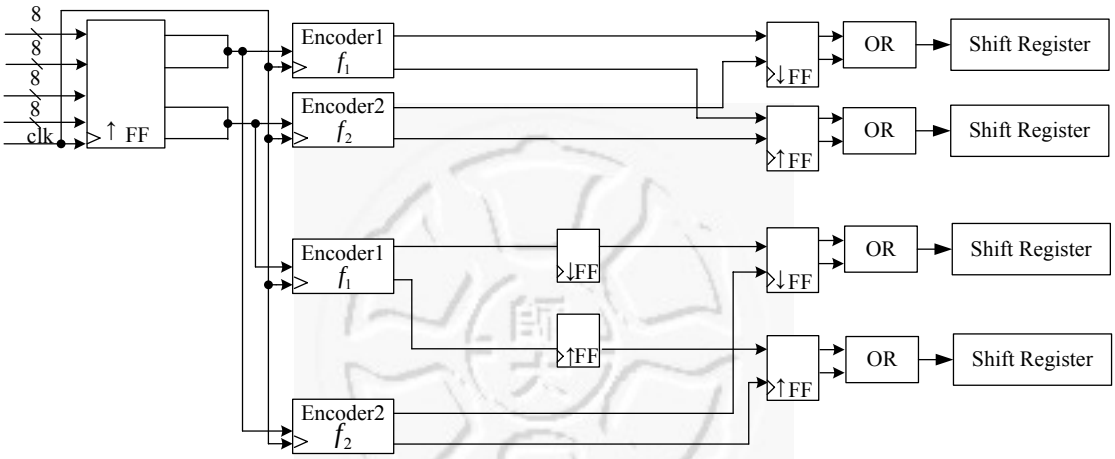

Figure 5.6: The partial encoding circuit with $q = 4$

$f_2(t_{n+2}, t_{n+3})$. Another full encoding function $f(t_{n-2}, t_{n-1}, t_n, t_{n+1})$ is also formed at this moment. $f_1(t_{n-2}, t_{n-1})$ and $f_2(t_n, t_{n+1})$ establish the full encoding function $f(t_{n-2}, t_{n-1}, t_n, t_{n+1})$ also by OR operation.

As the dataflow shown in the figure above, the alphabet size of four can be processed. In the next section, we will brief explain the experimental result.

5.3 Experimental Result

This section provides the experimental result of the partial encoding architecture. As illustrated in Table 5.1, the throughput, logic elements per character, memory bits and operating frequency are analyzed by the design of the symbol encoder without partial encoding and the design of the symbol encoder with partial encoding with $q = 2$. Moreover, both architecture adopt the bitmap encoding operation for the shift register. We can observed from the table that the design of symbol encoder without partial encoding performs well not only in the logic element per character but also in the throughput. The throughput of the design without utilizing the partial encoder can achieve 6.75 Gb/s which is better than

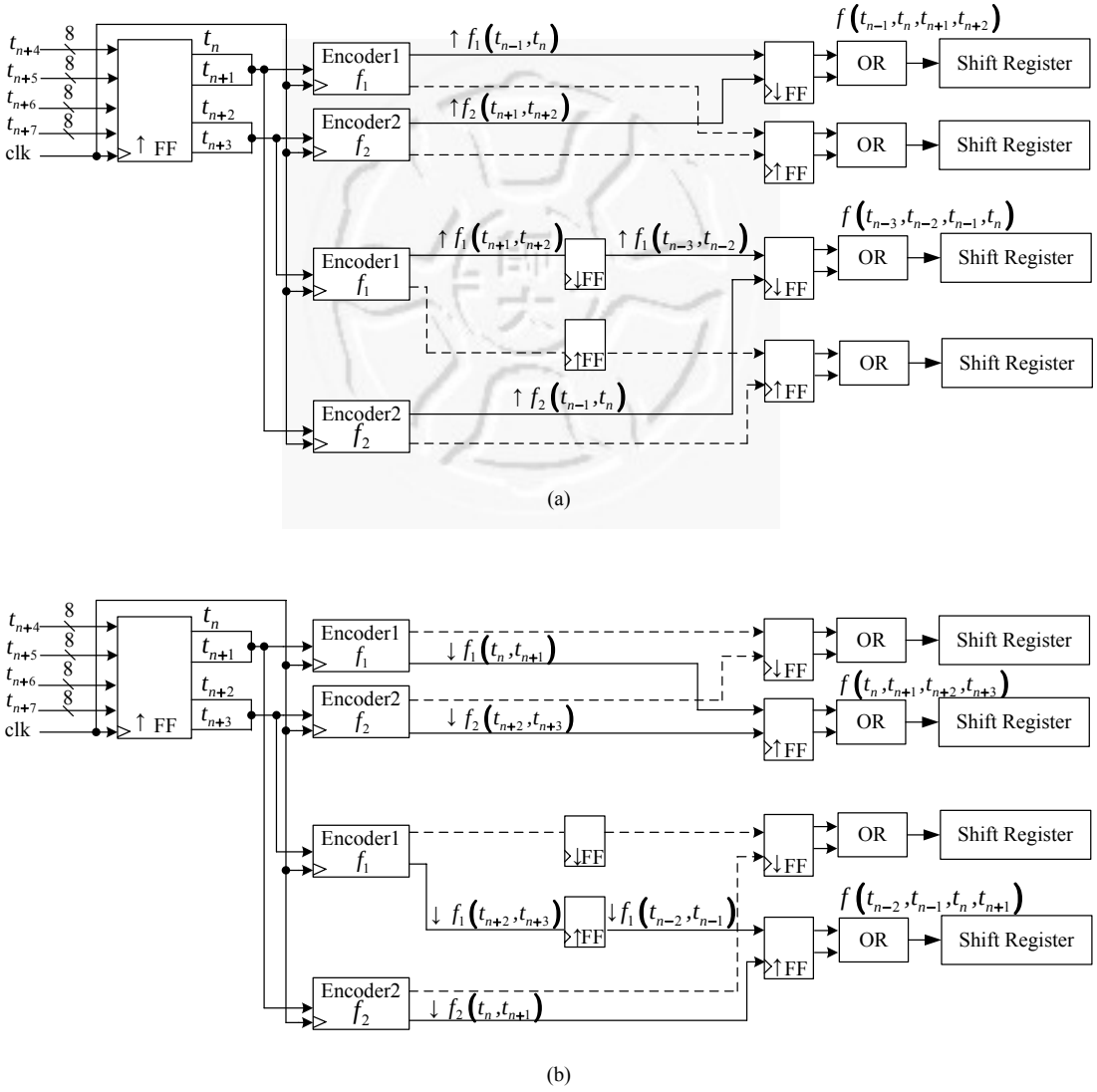

Figure 5.7: Dataflow of the partial encoding circuit with $q = 4$, (a) Rising edge (Obtain $f(t_{n-3}, t_{n-2}, t_{n-1}, t_n)$, $f(t_{n-1}, t_n, t_{n+1}, t_{n+2})$) (b) Falling edge (Obtain $f(t_{n-2}, t_{n-1}, t_n, t_{n+1})$, $f(t_n, t_{n+1}, t_{n+2}, t_{n+3})$)

Table 5.1: Comparisons of the bitmap encoding circuit with $q = 2$ between symbol encoder without partial encoding architecture and symbol encoder with partial encoding architecture, where the number of characters available for pattern matching is 1568 characters.

Design	Throughput (Gb/s)	LE /char	Memory bits	Operating Frequency (MHz)
Symbol Encoder Without Partial Encoding Architecture	6.75	0.7	0	422.12
Symbol Encoder With Partial Encoding Architecture	5.23	0.92	0	326.8

5.23 Gb/s of the design with utilizing the partial encoder. The logic element per character of the design without utilizing partial encoder only takes 0.7 LE per character which is still better than the design with utilizing the partial encoder of 0.92 LE per character. The statistics show us that the design of symbol encoder with partial encoding performs worse than the design of the symbol encoder without partial encoding on two characters per symbol at a time.

As shown in Table 5.2, we expand from two characters per symbol at a time with $q = 2$ to four characters per symbol at a time with $q = 4$. As the characters per symbol grow, all combination of the symbol increases. Hence, utilizing the symbol encoder without partial encoding will become a disaster not only in the logic elements per character but also in the throughput either. From the Table 5.2, the throughput of the design of symbol encoder with partial encoding architecture can attain 9.2 Gb/s which performs better than the design of the symbol encoder without partial encoding architecture with the throughput 5.92 Gb/s. The statistics of the logic element per character is obviously shown that the design utilizing partial encoding architecture only need 2.75 LE per character instead of 6.86 LE per character of the design without utilizing the partial encoding architecutre. Therefore, a truth has been proved that utilizing the partial encoding architecture performs much more better than not utilizing the partial

Table 5.2: Comparisons of the bitmap encoding circuit with $q = 4$ between symbol encoder without partial encoding architecture and symbol encoder with partial encoding architecture, where the number of characters available for pattern matching is 556 characters.

Design	Throughput (Gb/s)	LE /char	Memory bits	Operating Frequency (MHz)
Symbol Encoder Without Partial Encoding Architecture	5.92	6.86	0	184.95
Symbol Encoder With Partial Encoding Architecture	9.2	2.75	0	287.52

encoding architecture as the characters per symbol increase. The target FPGA devices for the implementation is Altera Stratix EP1S40.

The throughput, number of characters, logic elements per character, memory bits and operating frequency are analyzed by the ROM-based architecture, bitmap encoding architecture and partial encoding architecture as shown in Table 5.3.

Table 5.4 compares the FPGA implementations of the proposed architecture with the existing works. Notice that the exact comparisons of these circuits may be difficult since that different FPGA devies are utilized. However, it can still be observed from the table that our circuits have effective throughput and logic elements per character as compared with the existing work. The reason is that our design is based on the simple shift-or algorithm. The simplicity of the circuit allows the string matching operations to be performed at high clock rate with small area cost. The fact demostrated from the table shows the effectiveness of our design.

Table 5.3: Comparisons of ROM-based Architecture, Bitmap Encoding Architecture and Partial Encoding Architecture

Design	Throughput (Gb/s)	Chars	LE/ char	Memory bits	Operating Frequency(MHz)
ROM-based Architecture ($q = 1$)	1.94	8000	0.96	131,376	242.37
ROM-based Architecture without ROM sharing ($q = 2$)	5.14	1568	1.09	40,768	321.03
ROM-based Architecture with ROM sharing ($q = 2$)	4.65	1568	1.08	20,826	290.87
Bitmap Encoding Architecture ($q = 1$)	3.11	8000	0.53	0	388.35
Bitmap Encoding Architecture ($q = 2$)	6.75	1568	0.7	0	422.12
Bitmap Encoding Architecture ($q = 4$)	5.92	556	6.86	0	184.95
Partial Encoding Architecture ($q = 2$)	5.23	1568	0.92	0	326.8
Partial Encoding Architecture ($q = 4$)	9.2	556	2.75	0	287.52

Table 5.4: Comparisons of various string matching FPGA designs.

Design	Device	Throughput (Gb/s)	No. of chars	LE /char
Bitmap Encoding Architecture ($q = 1$)	Altera Stratix EP1S40	3.11	8000	0.53
Bitmap Encoding Architecture ($q = 2$)	Altera Stratix EP1S40	6.75	1568	0.7
Partial Encoding Architecture ($q = 4$)	Altera Stratix EP1S40	9.2	556	2.75
Baker-Prasanna (Tree) [4]	Xilinx Virtex2VP100-7	2.0	4518	0.42
Clark-Schimmel [5]	Xilinx Virtex2-8000	2.2	7996	1.88
Cho-MSmith [6]	Xilinx Spartan3-2000	3.2	6805	0.9
Hutchings et al. [7]	Xilinx Virtex-1000	0.248	8003	2.57
Gokhale et al. [8]	Xilinx VirtexE-1000	2.2	640	15.2
Moscola et al. [9]	Xilinx VirtexE-2000	1.18	420	19.4
Singaraju et al. [11]	Xilinx Virtex2VP30-7	6.41	1021	2.2
Sourdis-Pnevmatikatos [12]	Xilinx Spartan3-5000	4.91	18000	3.69