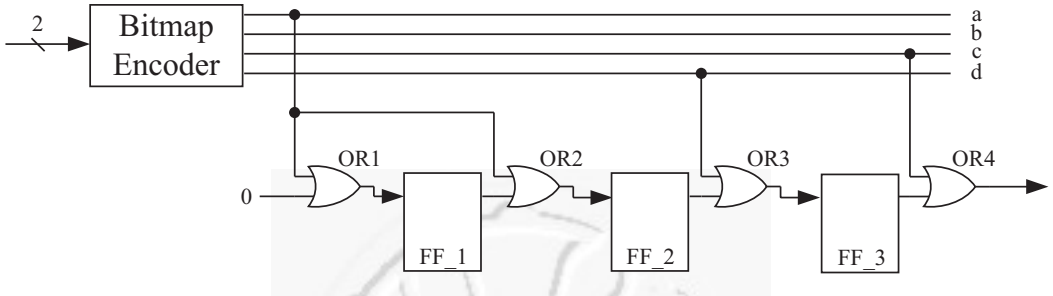
Chapter 4

Bitmap Encoding Architecture

Although the direct ROM implementation by the embedded memory bits is simple, it has plenty of drawbacks. The ROM may operate at slower speed as compared with the logic elements. For instance, the maximum clock rate of the embedded memory M4K of Altera Stratix is only 320 MHz; whereas, the maximum operating frequency of the logic elements of Altera Stratix is 420 MHz. Hence, the ROM implementation by the embedded memory bits may become the bottleneck of the system's throughput. Moreover, the same ROM cannot be shared by different rules. As the number of Snort rules grows, the consumption of embedded memory bits will be unbelievable high for the circuit. In this chapter, we propose a bitmap encoding architecture which uses only shift registers and bitmap encoder without any ROM for the hardware implementation.

Table 4.1: Table of the bitmap encoder

Input	Output
a	0111
b	1011
c	1101
d	1110



(a)

Input	Pattern 1
a	0011
b	1111
c	1110
d	1101

(b)

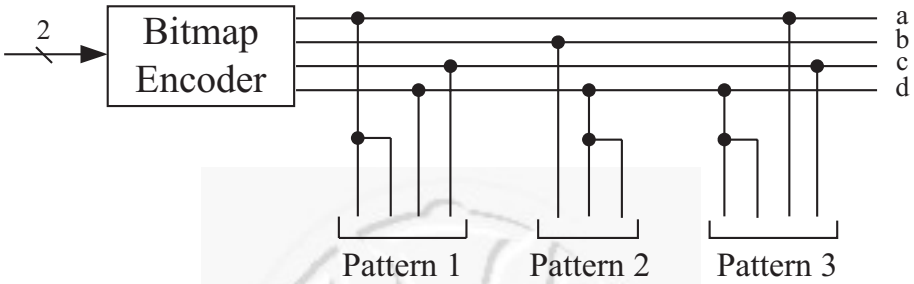
Figure 4.1: A simple example of the proposed circuit for the pattern *aadc* and the total symbol a, b, c, d, (a)The architecture (b)Table of the pattern

4.1 Basic Module Circuit

In the proposed circuit, the shift-or cirucit is implemented without ROMs. In consequence, the embedded memory bits are no more required, and higher operating frequency can be achieved. The ROMs are substituted by a simple bitmap encoder, which can be shared among different rules. In addition, the proposed circuit is well suited for the implementation of systems containing large number of Snort rules.

The bitmap encoding circuit is only comprised of the bitmap encoder and a shift register. The bitmap encoder possesses $|\Sigma|$ bits output. When the symbol s_k is the input to the bitmap encoder, only the k-th bit output will be set to zero. On the other perspective, the others are all set to one. Table 4.1 shows a simple example of the bitmap encoder for the alphabet $\Sigma = \{a, b, c, d\}$ with $s_1 = a$, $s_2 = b$, $s_3 = c$ and $s_4 = d$.

The shift register also contains $m - 1$ flip-flops and m OR gates. The bitmap encoding circuit is quite similar as the ROM-based circuit, where each OR gate



(a)

Input	Pattern 1	Pattern 2	Pattern 3
a	0011	111	1101
b	1111	011	1111
c	1110	110	1110
d	1101	101	0011

(b)

Figure 4.2: An example of three patterns (*aadc*, *bdd* and *ddac*) share the same bitmap encoder, (a)The architecture (b)Table of three patterns

contains two inputs. One of them is from the output bit of previous flip-flop. The second input is connected to the outputs of the bitmap encoder. The connection is depended on the i-th character p_i of the pattern. When the character p_i of the pattern is equivalent to the symbol s_k, we connect the second input of OR gate to the k-th output of the bitmap encoder. Based on eq.(2.2), we conclude that each OR gate i obtains $S_k[i]$ from the bitmap encoder when the input symbol is s_k. That is to say, the bitmap encoder can exactly replace the ROM for hardware shift-or implementation.

A simple example of the bitmap encoding circuit for pattern *aadc* and the total symbol a, b, c, d is demostrated in Figure 4.1, where the bitmap encoder is identical to that shown in Table 4.1. As illustrated in Figure 4.1(a), since p_1 and p_2 are equivalent to a, both the OR1 and OR2 are connected to the first output of the bitmap encoder. Next, we connect the OR3 and OR4 to the 4th and 3rd outputs of the bitmap encoder because that p_3 and p_4 are d and c respectively. Based on Table 4.1, it follows that when symbols a, b, c and d are

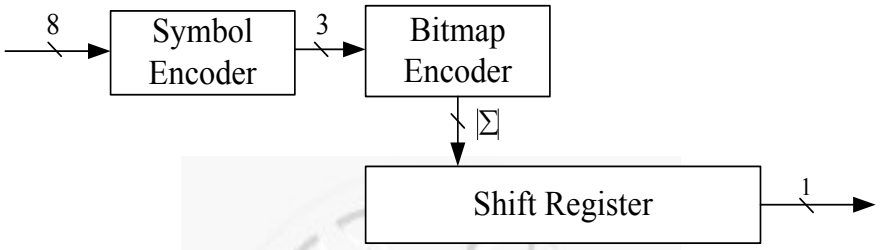


Figure 4.3: The increase of a symbol encoder for reducing the bitmap encoder size. In this example, each input charcter is assumbed to be an ASCII code (8 bits). We also assume the Snort rule uses only 7 symbols in the alphabet. The output of the symbol encoder is 3 bits.

the input symbols, the shift register receives the bit vectors 0011, 1111, 1110 and 1101, respectively. That is, when the input symbol is s_k, the shift register receives S_k from the bitmap encoder. Therefore, we can use the bitmap encoder to completely replace the ROM implementation.

One merit of utilizing the bitmap encoder is that the bitmap encoder is independent of the pattern. Only the connection between the bitmap encoder and the shift register is depended on the pattern. Therefore, different Snort rules can share the same bitmap encoder. An example of three patterns $aadc, bdd$ and $ddac$ sharing the same bitmap encoder is illustrated in Figure 4.2. The bitmap

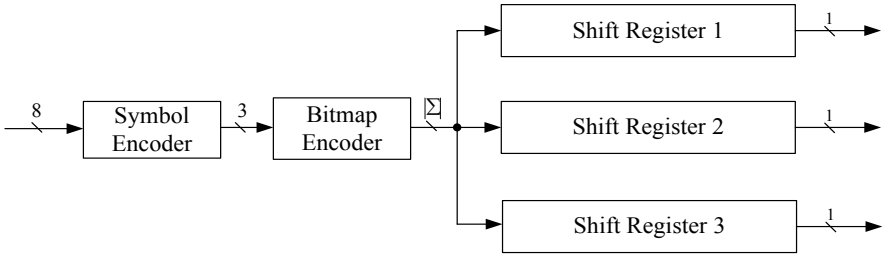


Figure 4.4: The sharing of the same symbol encoder and bitmap encoder by three different Snort rules. Each character is also assumed to be an ASCII. All the Snort rules use the same alphabet comprised of 7 symbols.

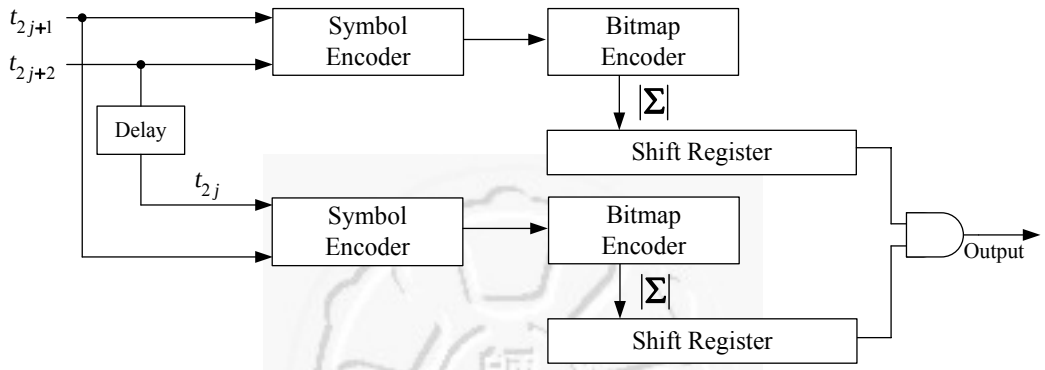


Figure 4.5: The structure of a high throughput module circuit processing two characters at a time ($q = 2$) with the bitmap encoder.

encoder has four outputs since that the total symbol is a, b, c, d. As shown in Figure 4.2(a), three patterns share the same bitmap encoder and the connection is depended on pattern. With the input symbol a, b, c and d, the shift register can acquire the bit vectors (0011, 1111, 1110, 1101) with pattern1, (111, 011, 110, 101) with pattern2, and (1101, 1111, 1110, 0011) with pattern3 as shown in Figure 4.2(c).

Similar as the ROM-based circuit, some symbols in the alphabet are not included in the Snort pattern. The area complexity can be further reduced that those symbols can share the same output in the bitmap encoder. Hence, we can incoporate a symbol encoder to map those unused symbols to the same output in the bitmap encoder as shown in Figure 4.3.

Although utilizing the symbol encoder can reduce the area cost of bitmap encoding circuit, it will not work well when rules have different sets of unused symbols. One approach to solve the problem is to divide the Snort rules into several groups. Moreover, the rules in each group use the same set of symbols. Therefore, all the rules in the same group can share the same symbol encoder and bitmap encoder as illustrated in Figure 4.4.

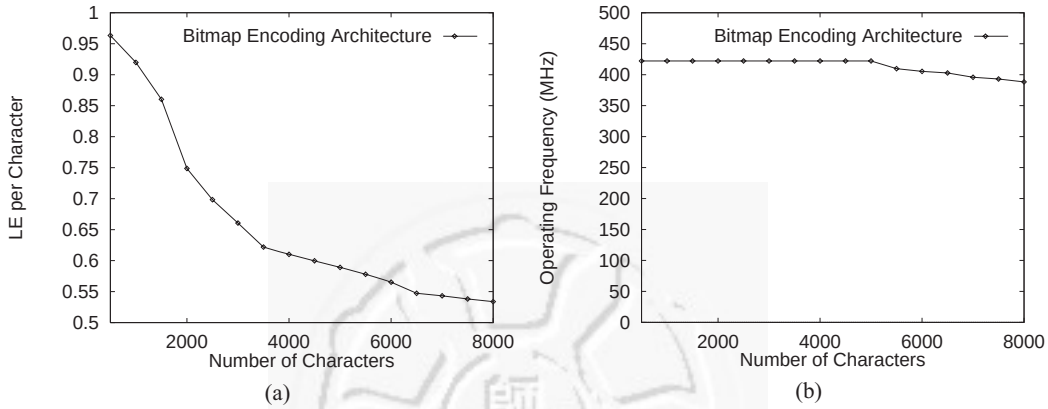


Figure 4.6: The performance of the proposed circuit with $q = 1$ for various rule sets sizes ranging from 500 characters to 8000 characters (a)LE per character (b)Operating Frequency.

Table 4.2: Comparisons of the proposed architecture with $q = 1$ and $q = 2$, where the number of characters available for pattern matching is 1568 characters.

Design	Throughput (Gb/s)	Logic cells /char	Memory bits	Operating Frequency (MHz)
$q = 1$	3.38	0.7	0	422.12
$q = 2$	6.75	0.7	0	422.12

4.2 High Throughput Module Circuit

The throughput of the NIDS can be improved further by processing q characters at a time. Actually, the high throughput module circuit in bitmap encoding architecture is the same as that in the ROM-based architecture illustrated in chapter 3.2. What makes the different is that the ROM is totally replaced by the bitmap encoder. Figure 4.5 shows the high throughput module circuit of bitmap encoding architecture.

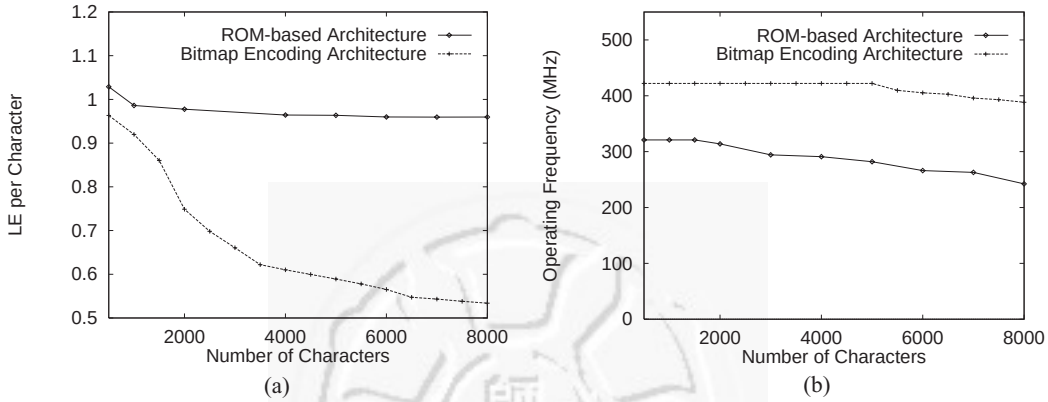


Figure 4.7: The performance of the ROM-based and bitmap encoding circuit with $q = 1$ for various rule sets sizes ranging from 500 characters to 8000 characters (a)LE per character (b)Operating Frequency.

4.3 Experimental Results

This section presents the experimental results of the proposed architecture for NIDS. Figure 4.6 demostrates the average number of LEs per character and operating frequency of the proposed circuit with $q = 1$ for various rule sets with sizes ranging from 500 characters to 8000 characters. In this experiment, both the symbol encoder and bitmap encoder are shared by different rules for reducing the area cost of the FPGA implementation. From Figure 4.6, we can observed that the operating frequency of the proposed circuit is stable over a wide range of the rule set sizes. Furthermore, the average number of LEs per characters suddenly decreases as the size of rule set increases. Particularly, the average number of characters becomes only 0.53 LE/char when the rule set size is 8000 characters.

As shown in Table 4.2, it compares the throughput, logic elements per character, memory bits and operating frequency of the proposed circuits with $q = 1$ and $q = 2$. Since the circuit with $q = 2$ processes two characters at a time, it has higher throughput than the circuit with $q = 1$ processing one character at a time. In addition, we can also observed from the table that no more ROM is required in the bitmap encoding architecture. Hence, the max frequency can break the bottleneck of the ROM-based circuit to achieve 422.12 MHz.

Figure 4.7 demostrates the comparisons between ROM-based circuit and bitmap based circuit with $q = 1$ for various rule sets sizes from 500 characters to 8000 characters. As shown in Figure 4.7(a), the logic element per character of the bitmap encoding circuit has greatly reduced as compared with that of the ROM-based circuit. The reason is that Altera Quartus II employ a technique to share some flip-flops among all the rule sets to reduce the consumption of logic elements. Frome Figure 4.7(b), we can observed that the operating frequency of the bitmap encoding circuit start to drop from 422.12 MHz. Instead of the bottleneck from the ROM-based architecture, the bitmap encoding architecture is therefore an effective approach to achieve both high throughput and low area cost.