Chapter 3
Custom automata system for Snort rules

General automata may process Snort rules slowly because those automata take only one alphabet in transition action when matching. Another problem is common automata couldn't handle pattern-relationships. It is not a good idea in performance to handle all pattern relationships and PCRE patterns with CPU computing after finishing pattern-matching. It may cost extra overheads while processing pattern relationships and PCRE patterns because general pattern-matching automata couldn't handle them and CPU has to do extra computing for them. Thus an automata system is designed in this thesis which is suitable not only to process Snort rules and pattern relationships but also to be implemented in hardware.

We put PCRE data structure into AC algorithm to process Perl Compatible Regular Expression. Then common exactly pattern-matching and pattern-relationship matching are handled by AC and PCRE pattern-matching is processed by PCRE data structure.

At first, translate and divide every Snort rule into our specific header and content rules. According to those specific rules, construct our header and content automata. Then put header and content automata into on-chip memory in hardware as our hardware design and finally it could only spend $O(L)$ to check all rules in those two automata with parallel matching-engine in hardware.

3.1 Advantages in custom automata system

There are three advantages in this automata system.

1. **Scalability**: With Regular Expression and custom automata, it could handle numbers of header types and content patterns in Snort rules for packet classification or intrusion detection.
2. **Flexibility and Specification**: It supports almost all Snort rules descriptions, including header types, content patterns, pattern-relationships and PCRE patterns, even specific types of rules.
3. **Less Storage Space and Better Performance**: Our automata are different from normal one for general purpose of pattern-matching because we may optimize it for Snort rule-matching to save space and raise speed in hardware design.

3.2 The architecture in custom automata system

The architecture of our custom automata system is divided into header and content automata. Both of those two automata have two phases: Compiling and Matching.

Compiling:

While input Snort rules, two parser programs written in Perl first process them and translate rules into header simple rules and content simple rules in the specific format. Then our header program and content program could handle those simple rules and create our custom header and content automata, as shown in Figure 3-1.

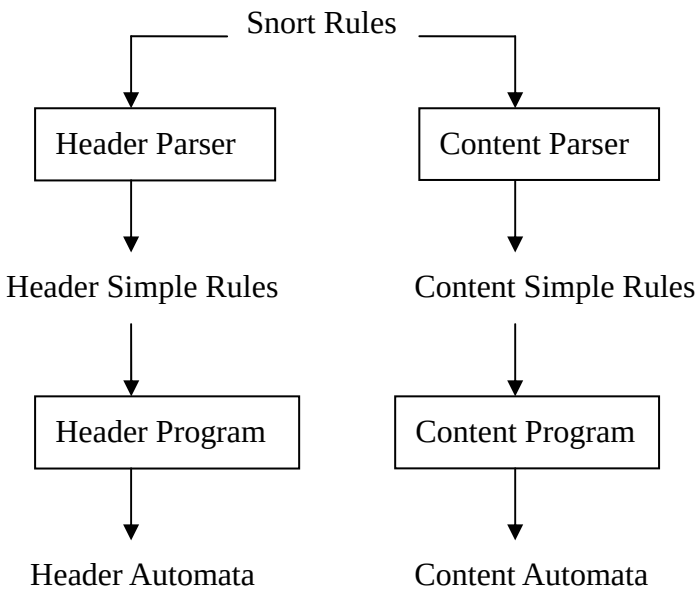


Figure 3-1 Overview of custom automata system for Snort rules when compiling

Matching:

After finishing compiling, we have already created our header and content automata. When a packet entered our system, pre-processor first divides this packet into header specific data, which are those necessary header fields that we need when doing matching in header automata, and payload. Then header and content automata do matching and output matched rule IDs. Finally matched header and content rule IDs enter AND result unit to get whole matched rule IDs, as shown in Figure 3-2.

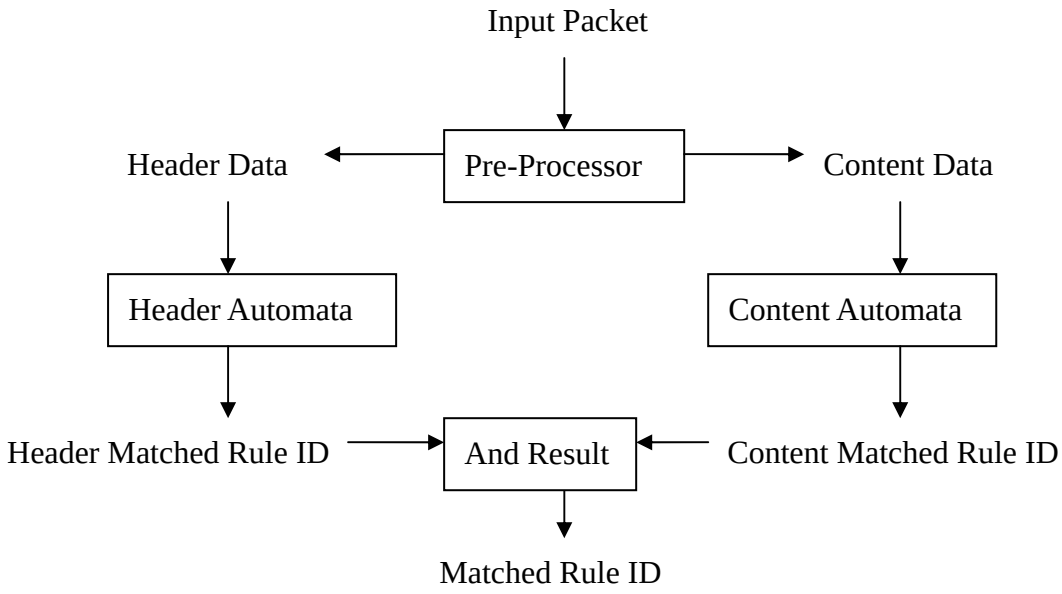


Figure 3-2 Overview of custom automata system for Snort rules when matching

The header and content parser could handle those types in Snort rules.

◆ **Header Parser:**

- Supported keywords:
  - IP: tos, dsize, id, ttl, ip_proto
  - TCP: seq, ack, window
  - Icmp: itype, icode, icmp_id, icmp_seq
  - Source_ip, Destination_ip, Source_port, Destination_port, protocol

Then header parser translates Snort rules into header simple rules.

- Header simple rule format:
  - Sid: tos dsize id ttl ip_proto….option:(seq ack window) (itype icode icmp_id icmp_seq)

All the header fields in header simple rules may be a constant value, not equal to a constant value, larger than a constant value, less than a constant value, a range of values, and any values expressed in "**.**"

◆ **Content Parser:**

- Supported keyword: Sid, content:, content:!, nocase, depth, offset, distance, within, pcre:, pcre:!

Then content parser translates Snort rules into content simple rules.

- Content simple rule format:
  - Sid: # pattern# # pattern# ….

All the patterns in content simple rules may be a content string, a PCRE string, or a pattern-relationship.

3.3 Supporting Regular Expression

In Regular Expression, there are lots of operations we have to support in EGREP and PCRE. The operations that our automata supports are listed below.

◆ **Beginning of Line(^)**, **End of Line($)**

Example: "**^abc**" means the first three characters in the beginning of this line are "abc".

◆ **Any(.)**, **Any Of ([])**, **Any But ([^])**

Example: "**.[abc][^123]**" means the first character could be anything, the second one must be the one of "a", "b", or "c", and the third one should not be any of "1", "2", or "3".

◆ **Branch(|)**

Example: "**ab|c**" means the first character is "a" and the second one could be "b" or "c".

◆ **Star(*)**, **Plus(+)**, **Question Mark(?)**

Example: "**a*b+c?**" means we could match nothing or more than one time of "a", match more than one time of "b", and match nothing or one "c".

- **Range(-)**

  Example: "**[2-6][d-f]**" means the first character should be the number in the range of "2" to "6" and the second one should be the "d", "e", or "f".

- **Exactly Match**

  Those are the basic operations in Regular Expression, but if we want to solve some relationships between patterns, we still need some solutions to these operations.

- **Times((Num))**

  Example: "**a(2,4)**" means the times that we have to match "a" is between 2 and 4

3.4 Implement header and content custom automata

Header Automata:

Unlike traditional AC automata, the header automata uses tree structure [15]. Each level has the unique header field to handle (eg: level 2 handles tos, level 3 handles dsize). If there exists a node with '**.**', that means this rule has no header pattern in this header field, equivalent to "don't care". This custom header could handle Snort header rules with performance while doing multi-pattern matching.

If there are these rules, header automata is constructed as shown in Figure 3-3.

R1: tos = 2, dsize != 3, id = 567

R2: tos = 2, dsize = 1

R3: dsize > 4

R4: dsize < 5

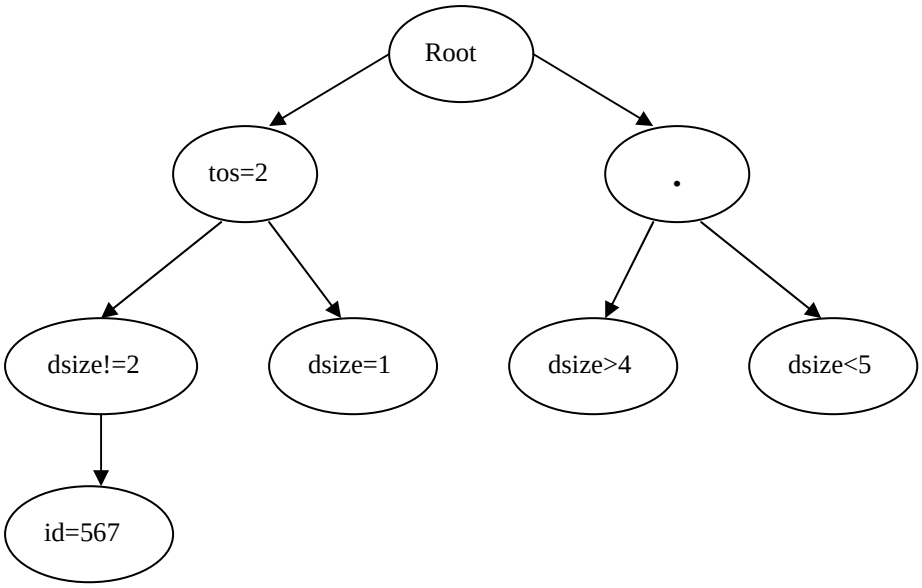


Figure 3-3 An example of custom header automata.

Content Automata:

The custom automata has many differences from common automata. First, in general automata, like AC, there is only one alphabet of patterns in every node. Now every node in content custom automata has the whole content string of this pattern. That means while doing matching, we view every pattern as one unit. This evolution may make performance better and make it easier to process the relationships between patterns. Second, cancel the parts of failure path in AC automata and use parallel matching. In matching phase, add all the nodes that will be checked now into a queue (now_chain) and wait for parallel matching. After finding one node matched, add child nodes into waiting-queue (next_chain) for the next step of matching.

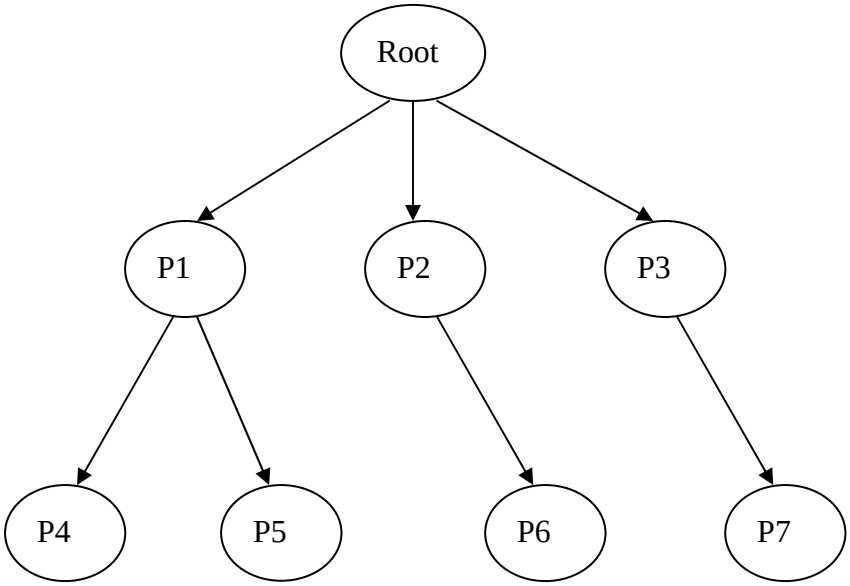


Figure 3-4 An example of custom content automata

For example, consider a content automata is built like the one shown in Figure 3-4. Add P1, P2, and P3 in level 1 into now_chain initially. If P1 and P2 are matched, add P4, P5 and P6 in level 2 into next_chain. After finishing all the nodes in now_chain, put all the nodes from next_chain to now_chain for the next matching and clean next_chain.

This implementation seems make performance worse than the original automata using AC algorithm with failure path. But it has much smaller size than Snort system even the software simulation of custom automata system doesn't doing parallel matching. The low storage requirements make it possible to put the whole automata into SRAM. With hardware parallel match engines as our design, it could achieve the goal to create a nice IDS in hardware with the performance of multi-pattern matching algorithm and lower storage requirements. Especially, it handles pattern relationships in the nodes of automata, instead of processing them in another units or using CPU to compute.

If there is a Snort rule R = ”content: abcd; content: xyz; distance: 4; within: 8;”, that means while matching “abcd” in position A of payload, we have to found “xyz” in position A+4~ A+8. Custom content automata record those relationships into the node of “xyz”. After matching the node of “abcd”, we could know the range of searching the pattern “xyz” and have ability to process those relationships.

All the nodes are divided into the following types.

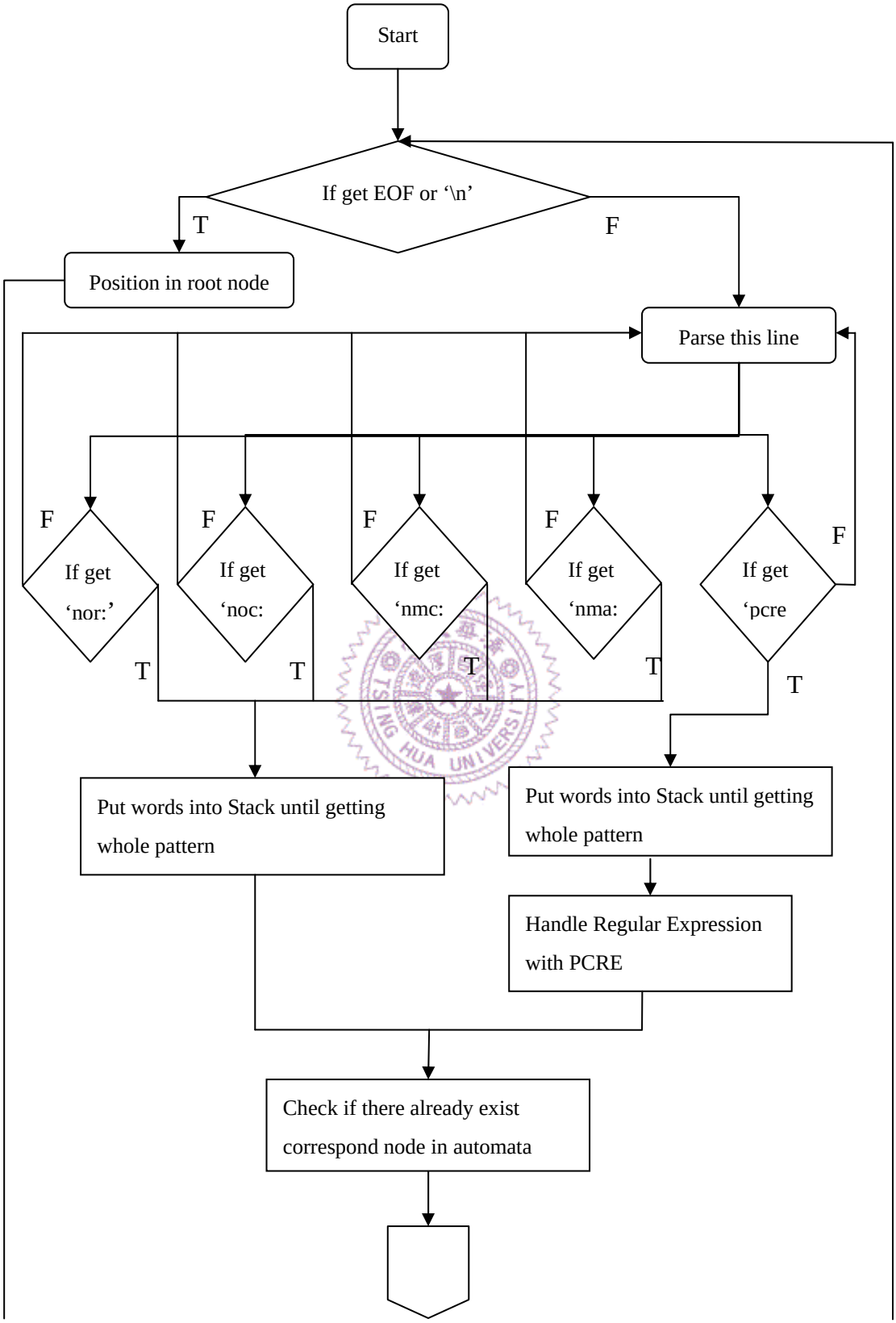
- Pattern node:
  - \1:match this normal content pattern
  - \2:match this pattern no matter its case(no case)
  - \3:don’t match this pattern and no matter its case(no case)
  - \4:don’t match this pattern
  - \0:match PCRE pattern
  - \5:don’t match this PCRE pattern

Also some operation information is added into a node:

- Operations:
  - Distance: depend on location of previous matched pattern
  - Within: depend on location of previous matched pattern
  - Offset: from the beginning of payload
  - Depth: from the beginning of the searching range

The flowcharts of our custom content automata while compiling and matching are illustrated in Figure 3-5 and Figure 3-6, respectively.

- **Compiling Steps**:



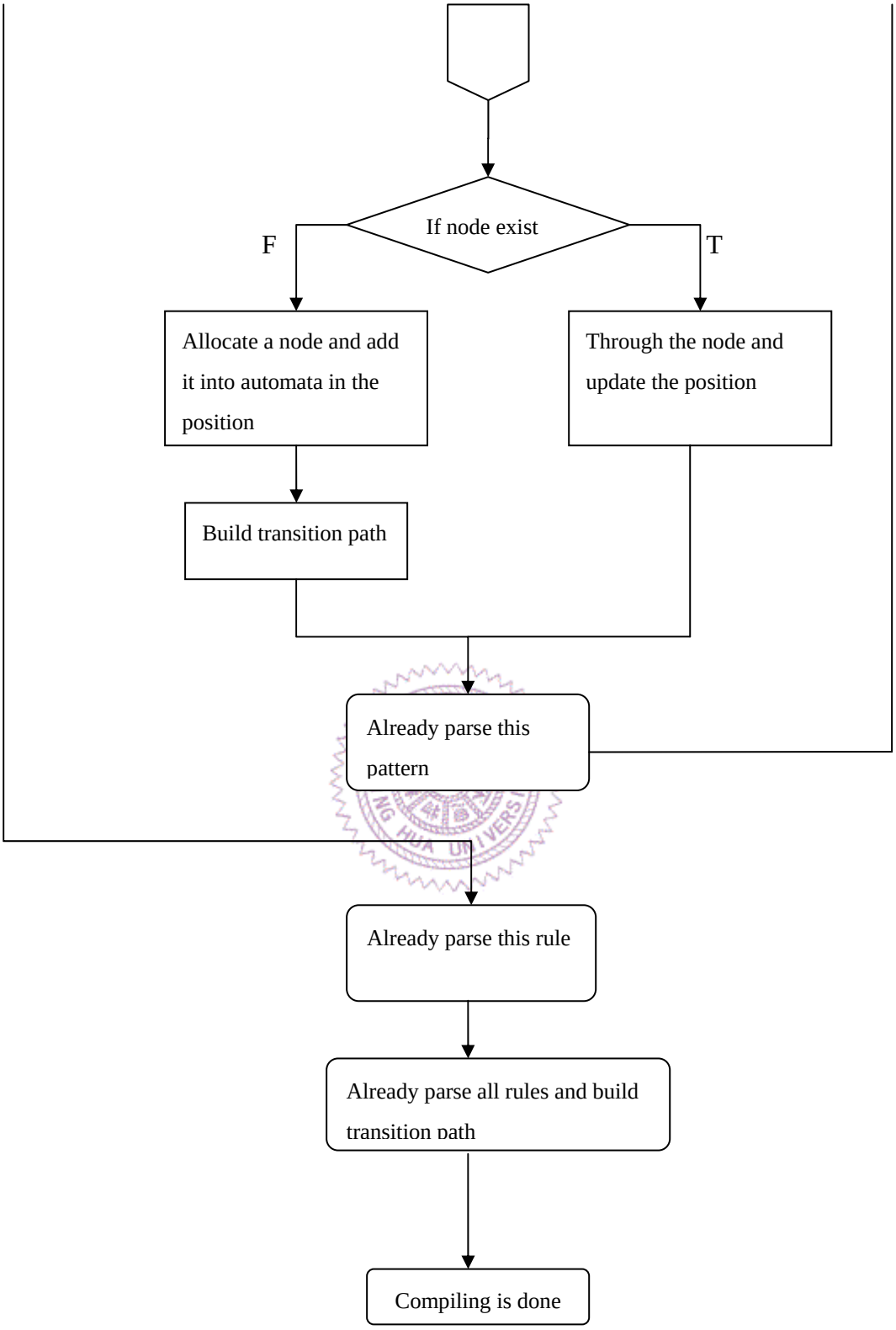
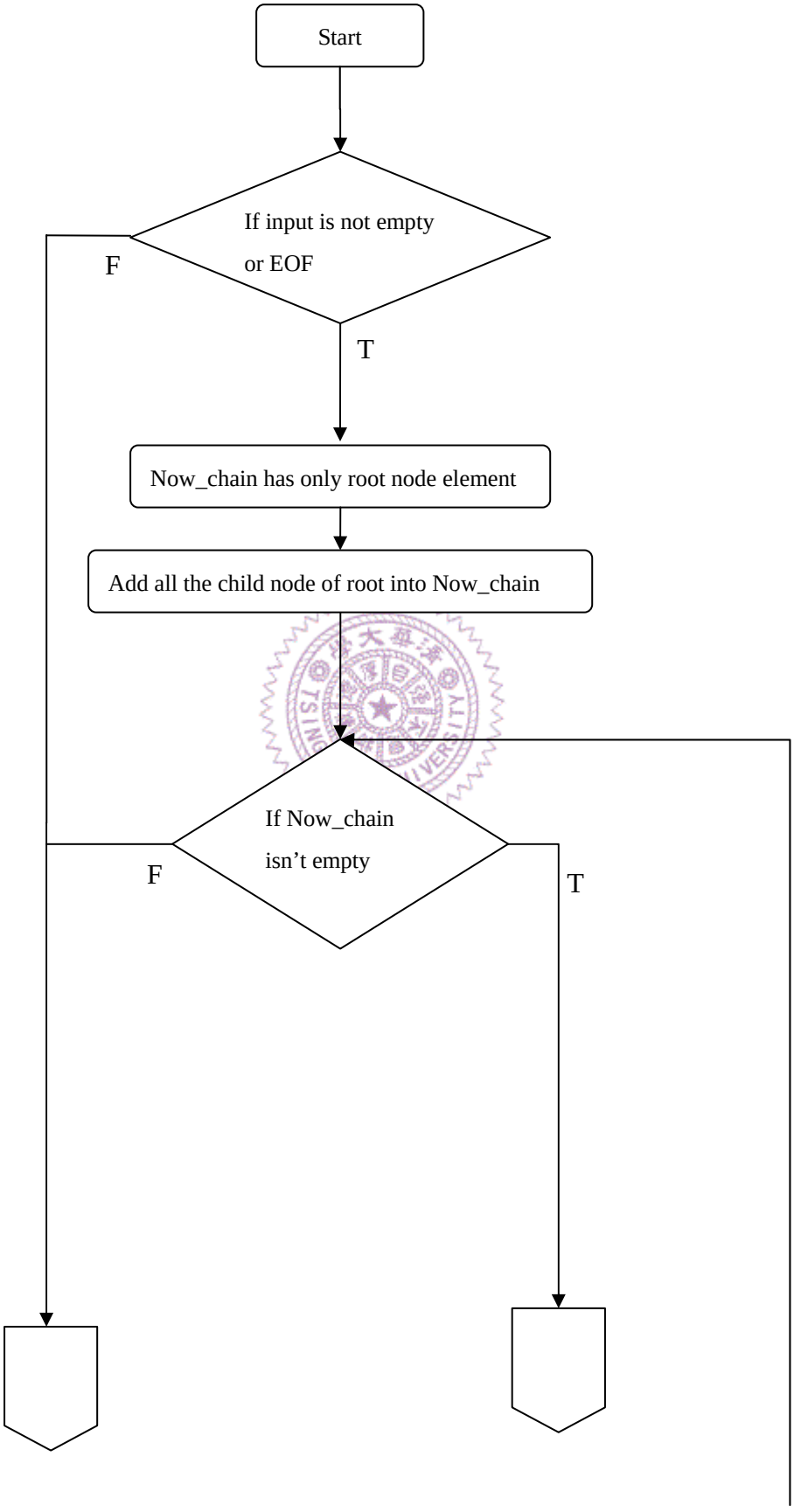


Figure 3-5 Compiling flowchart of custom content automata.

- **Matching Steps**:



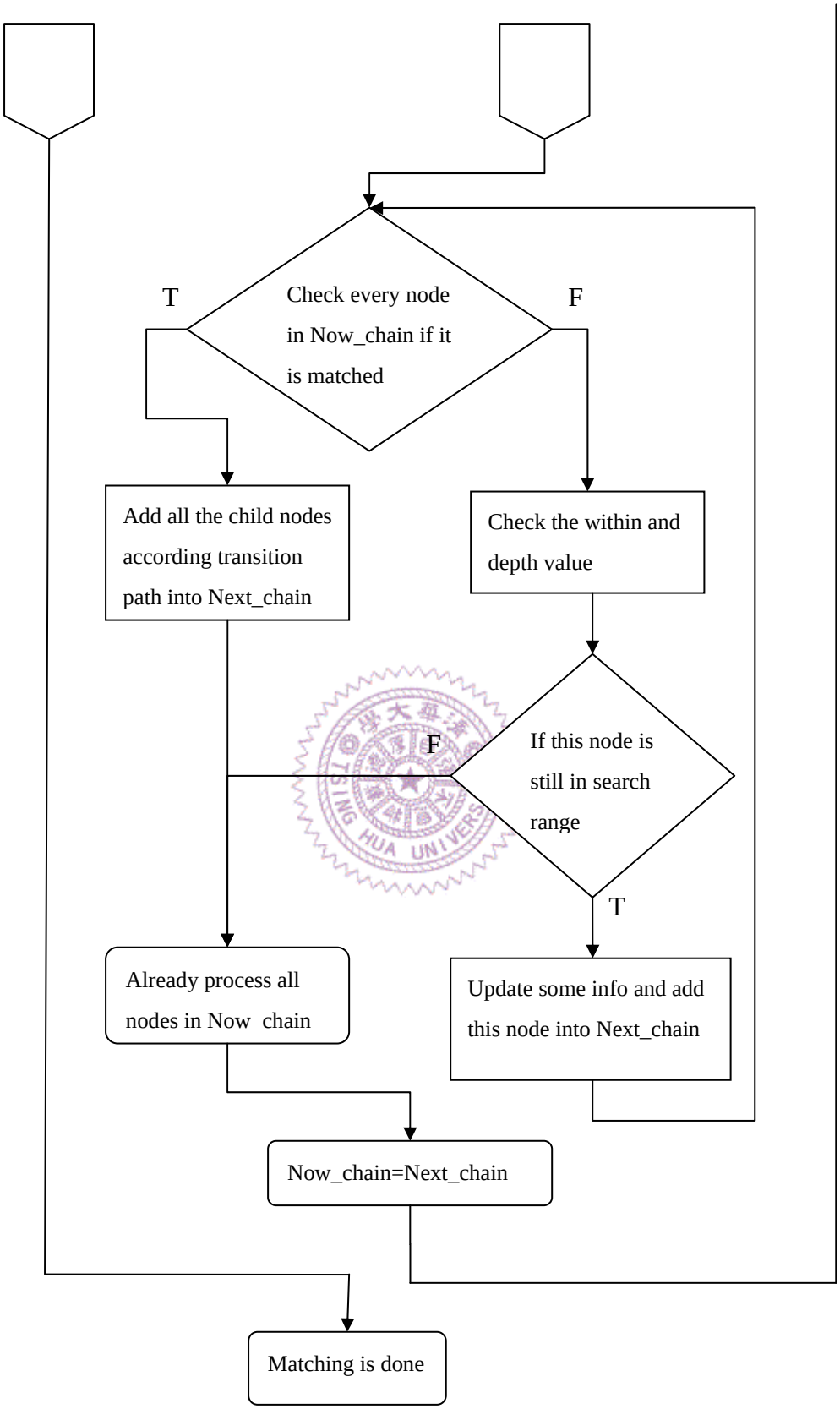


Figure 3-6 Matching flowchart of custom content automata.

For example, Figure 3-7 shows a rule and the corresponding constructed content automata.

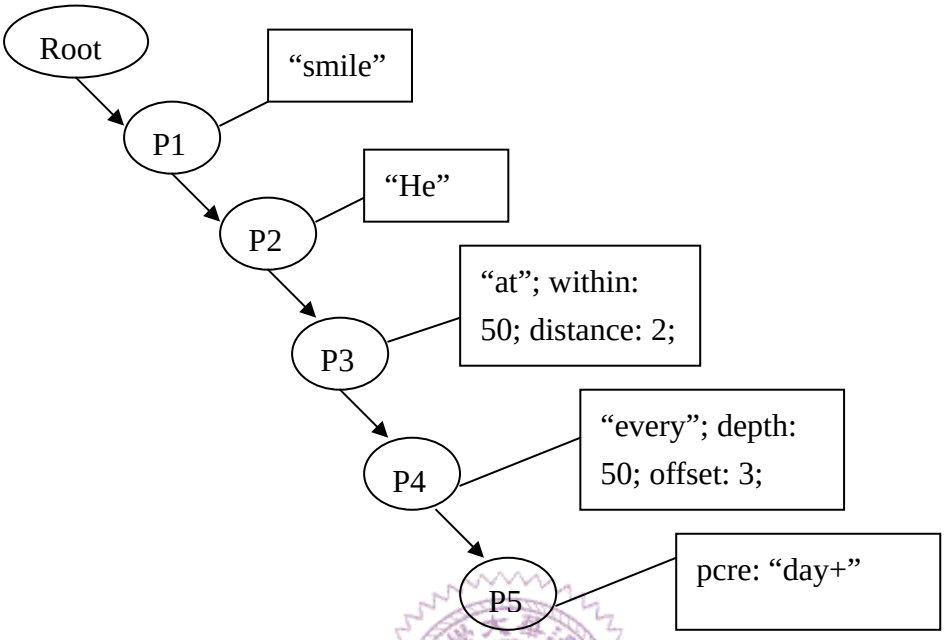


Figure 3-7 An example of custom content automata while matching.

If input string is "He smiles at me every dayyyy", first add P1 node into now_chain and search "smile" at location 0. It's unmatched so add P1 node into next_chain and search "smile" at location 1 and so on. Now find "smile" at location 3. Then add P2 into next_chain and search "He" at location 0 repeatedly. Then find "He" at location 0 and add P3 node into next_chain. Search "at" at location 2 and find it at location 10. Then add P4 into next_chain and search "every" at location 3. Find it at location 16 and add P5 into next_chain. Search Regular Expression "day+" at location 0 and find it at location 22.

3.5 Experiments and comparisons between custom automata system and Snort

We compare our custom automata system with Snort rules and Snort system. First we test the executing time in matching phase while input different lengths of packets. Testing CPU speed is 400 MHz and measurement unit is time tick (1/ CLK per second). The measure results are shown in Table 2 and Figure 3-8. We can see that the matching time is *O (L)* when input packet is length of *L*. Then we test the matching tine when input different lengths of packets which consist of normal alphabets or special strings matched Regular Expression rules. Testing CPU speed is 1.8 GHz and measurement unit is second.

Table 2 Matching time of custom automata system when input different data sizes

Input data size (bytes)	Executing matching time (time ticks)
64	741
100	1162
200	2804
300	3395
400	4566
500	5918
600	6870
700	7911
800	9214
900	10064

1000	11266
1100	12398
1200	13529
1300	14721
1400	15743
1500	16975

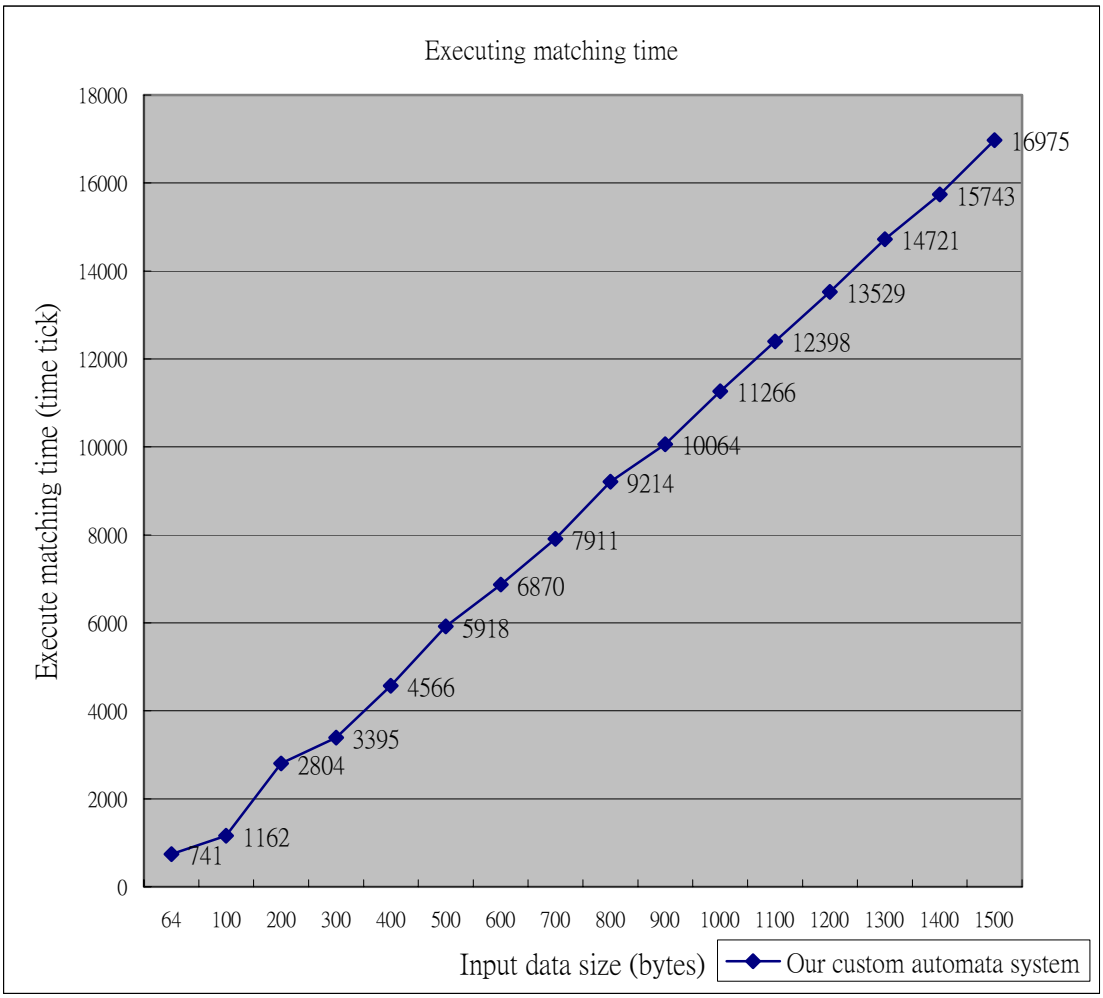


Figure 3-8 Matching time of custom automata system when input different data sizes.

The matching time of Snort system for different packets and that for our proposed custom automata system are shown in Table 3 and Table 4, respectively. We can also see that the data size consumed by our custom automata system is much smaller than that of Snort system. But the performance of our custom automata is worse than that of Snort system. Nevertheless, the software simulation acts like a single pattern matching algorithm when comparing a pattern in a node. It only compares one pattern step by step while doing matching and does not have any advantages of multiple pattern matching. To overcome this problem, putting our custom automata in SRAM and implement the whole system in our hardware design is a good solution to achieve multiple pattern matching. The hardware design for the proposed custom automata system is discussed in the next section.

Table 3 Matching time of Snort system for different packets

Using AC-std algorithm		user_time (sec)		memory
length (Bytes)	compile time (sec)	Content packet	PCRE packet	268.57 MB
100*1500	2.64 ~ 2.82	0.17	0.19	
100*1000		0.154	0.179	
100*500		0.135	0.155	
100*100		0.12	0.135	
Using AC-full algorithm		user_time (sec)		memory
length (Bytes)	compile time (sec)	Content packet	PCRE packet	139.52 MB / 271.5 MB if using 4B state
100*1500	18 .79~ 19.1	0.17	0.193	
Using SFKTrie (lowmem)		user_time (sec)		memory
length (Bytes)	compile time (sec)	Content packet	PCRE packet	16.87 MB
100*1500	0.11	0.244	0.274	
100*1000		0.21	0.23	
100*500		0.165	0.185	
100*100		0.128	0.149	
Using Modified Wu-Manber		user_time (sec)		memory
length (Bytes)	compile time (sec)	content packet	PCRE packet	51.28MB
100*1500	0.195	0.166	0.194	
100*1000		0.158	0.18	
100*500		0.138	0.157	
100*100		0.128	0.14	

Table 4 Matching time of custom automata system for different packets

system	user_time (sec)		memory
length (Bytes)	content packet	PCRE packet	361.538 KB
5000	2.28	2.34	
4000	2.06	2.08	
3000	1.4	1.4	
1500	0.7	0.71	
1000	0.48	0.49	
500	0.27	0.28	
100	0.11	0.12	

3.6 Implementation in hardware

According to the experiments mentioned above, the software simulation of the proposed custom automata system has smaller data size but worse performance. The main factor is that this simulation doesn't do parallel matching in software so all the pattern-matching jobs are similar as single pattern matching. However, the much less storage requirements are the most important advantage to implement in hardware. Although AC algorithm is multi-pattern matching, it is not easy to implement in hardware parallel architecture because AC algorithm only depends on failure path to achieve multi-pattern matching and its data size is too big indeed. Thus designing an efficiently hardware architecture for the custom automata system will overcome the performance concern in software simulation. The hardware architecture designed in this thesis is illustrated in Figure 3-9.

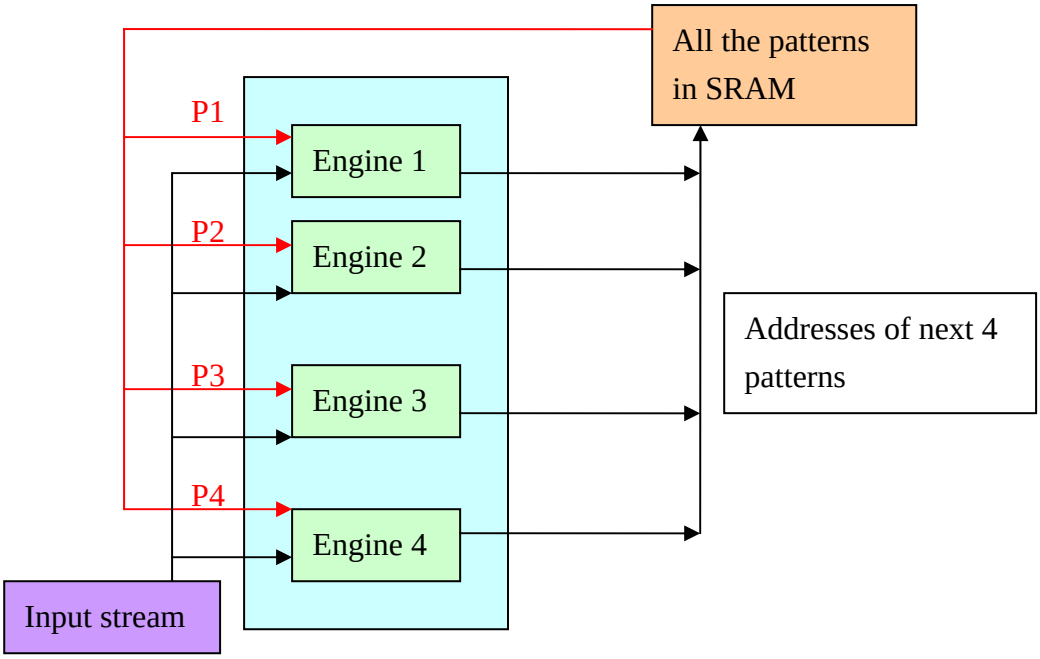


Figure 3-9 Hardware architecture for parallel pattern matching.

In our hardware design of custom automata system, the whole system is partitioned into four parts, as shown in Figure 3-10.

- **Hash Filter**: Using *M* hash functions to filter those pattern nodes which may be matched input payload in next matching step of matching engine.
- **Automata in SRAM**: Store constructed automata and load patterns into matching engine for next matching step
- **Matching Engine**: Use *N* different *P*-bit ALUs to handle matching processes. Let *N*P* be always greater than the length of every pattern in order to do multi-pattern matching at the same time. (in our assumption, *P*=32, but the value can be changed if necessary for seeking higher performance)
- **Regular Expression Engine**: Handle Regular Expression matching.

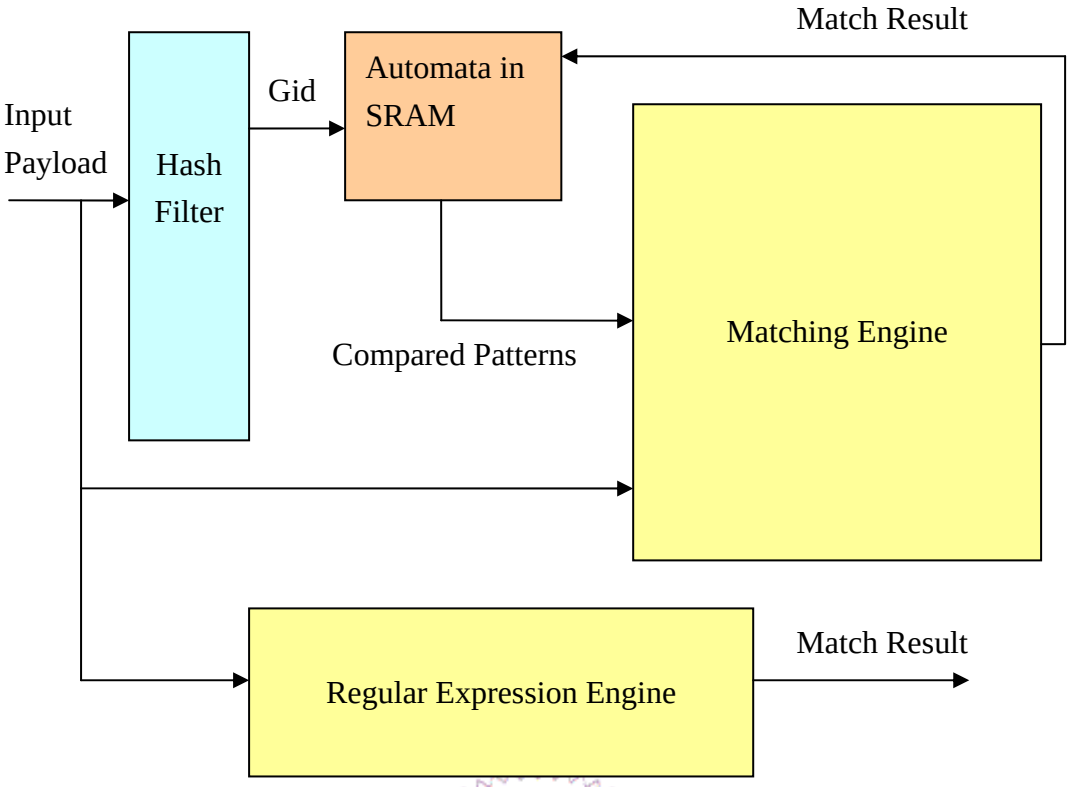


Figure 3-10 Hardware Design of Custom Automata System

- **Hash Filter**

In this hash filter, *M* different hash functions are used to calculate every pattern in the node of automata. In other words, we give every node an *M*-dimension attribute set. With different hash functions, all the nodes are divided into groups and have G_1, G_2, G_3,…,G_M hash values in *M*-dimension. The hash filter maintains a hash table stores those hash values of every node. When doing matching, the hash filter first takes every four bytes of input payload and uses *M* different hash functions to get *M* hash values. According to those hash values, search for the correspond nodes in the hash table. Those nodes whose hash values are the same as the input hash values may be matched in the next matching step of matching engine. And those nodes whose hash values are different from the input hash values have no possibility to be matched. With this hash filter, it only needs to check the nodes which may be matched instead

of checking all the nodes in matching engine. Therefore this hash filter could save our time and raise the performance of matching process.

For illustration, consider the example shown in Figure 3-11. Assume M = 2, the hash filter divides all the nodes in automata into two groups G_1 = 1 and G_1 = 2 with hash function 1 and three groups G_2 = 1, G_2 = 2, and G_2 = 3 with hash function 2. If input hash values = (2,2), then P7 and P8 may match this input and other nodes are never to match this input.

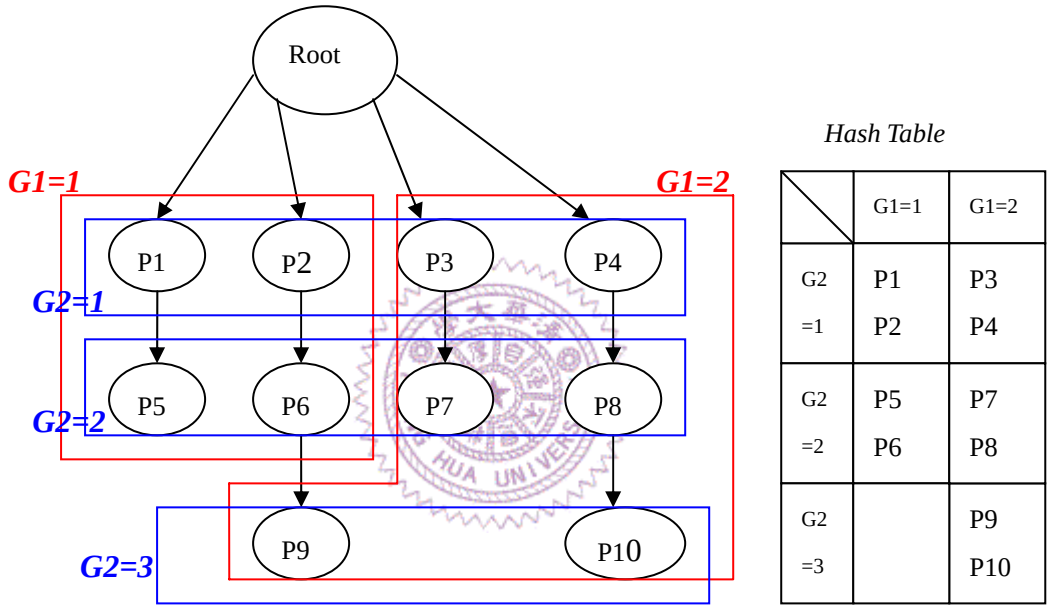


Figure 3-11 An example of group nodes with hash filter (M=2)

- **Automata in SRAM**

From the result of experiments in previous section, the data size produced of our custom automata system is much smaller than that of Snort. We could put all the automata into a SRAM to accelerate and save the memory-access time while reading patterns to match. After hash filter picks the correspond groups of nodes which may be matched, system will get those nodes in automata according to the group ID of hash table and input those patterns to matching engine for the next step of matching.

- **Matching Engine**

It has been pointed out that the length of every pattern in Snort rules is almost around 4~15 bytes [16]. In order to fit our hardware design, the matching method of our custom automata is then changed. In the original design, the whole pattern is matched byte by byte. In the new design, every four-byte substring is matched in a 32-bit ALU. For each pattern longer than four bytes, divide it into a sequence of 4-byte substrings, except the last one, which may be shorter than or equal to four bytes. And every substring has relationship “distance = 0”. Figure 3-12 shows an example to partition the pattern “abcdefghij” into three substrings “abcd”, “efgh”, and “ij”, and the relationship is “distance = 0”. Then the matching engine with *N* 32-bit ALUs is used to do the matching process in parallel. This design could match one pattern in only one step no matter of its length and the performance can be improved dramatically.

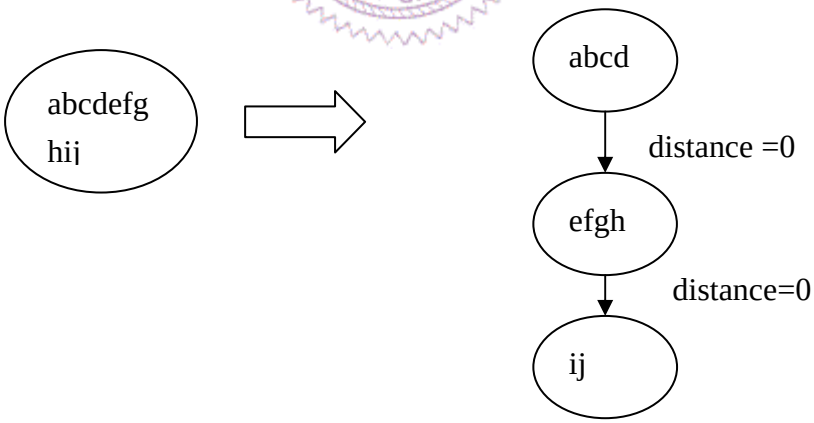


Figure 3-12 An example of dividing every pattern into 4-byte substrings.

The matching engine in hardware may focus on accelerating with using *N* 32-bit ALUs to check if every four bytes of input payload match any patterns. For example, a matching engine with *N* = 4 is illustrated in Figure 3-13. Four ALUs will be

employed to process a pattern with length of 16 bytes or more. For shorter patterns, only less number of ALUs are required and some of the ALUs may be arranged to process other patterns.

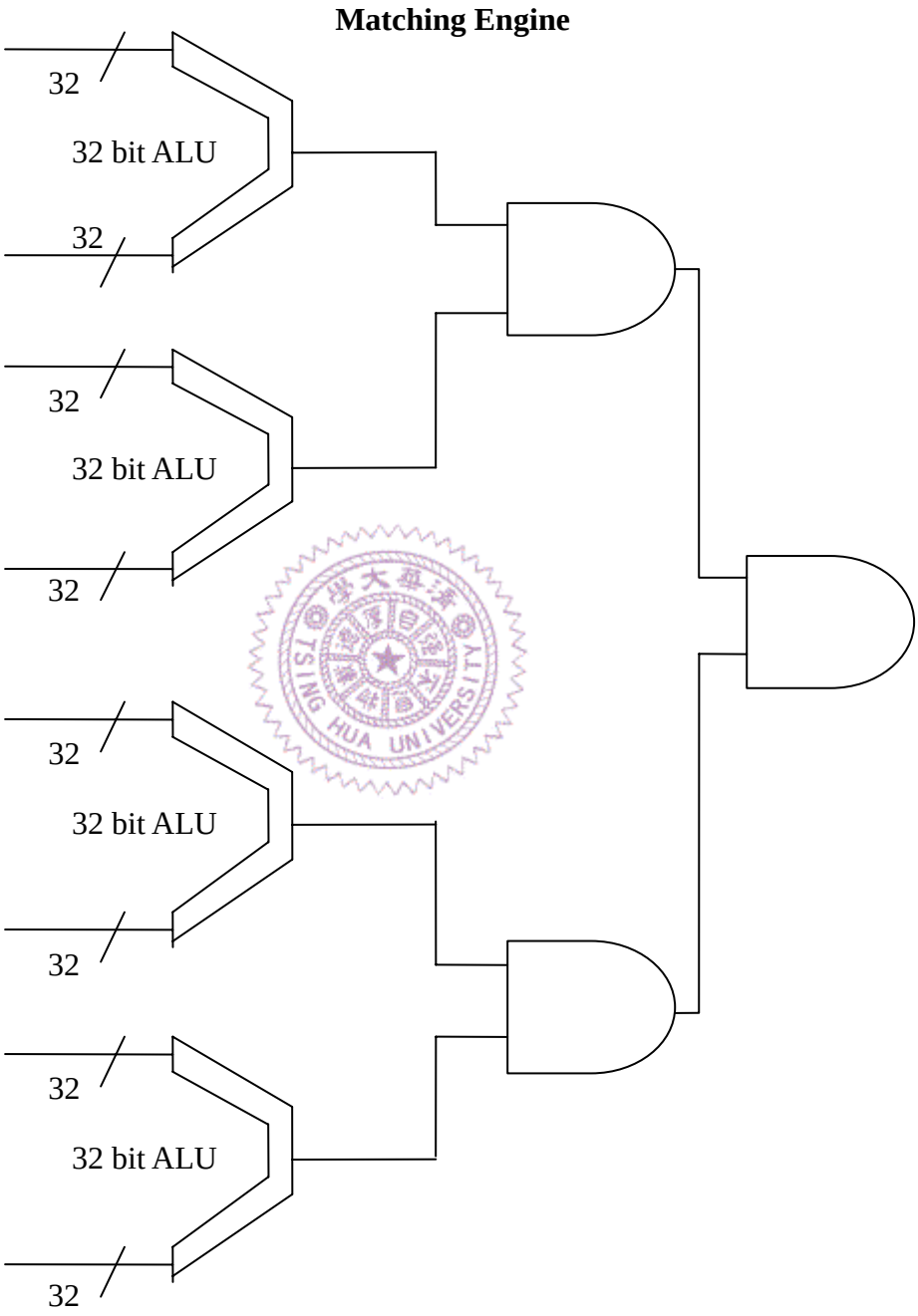


Figure 3-13 Matching engine in hardware with *N* 32-bit ALUs (*N*=4)

With the parallel ALUs to handle matching process, our system in hardware could achieve the performance of multi-pattern matching system. After adding a hash

filter in front of the matching engine, it only needs to pass the possible matched patterns into matching engine instead of all patterns. The matching engine will become more efficient and faster.

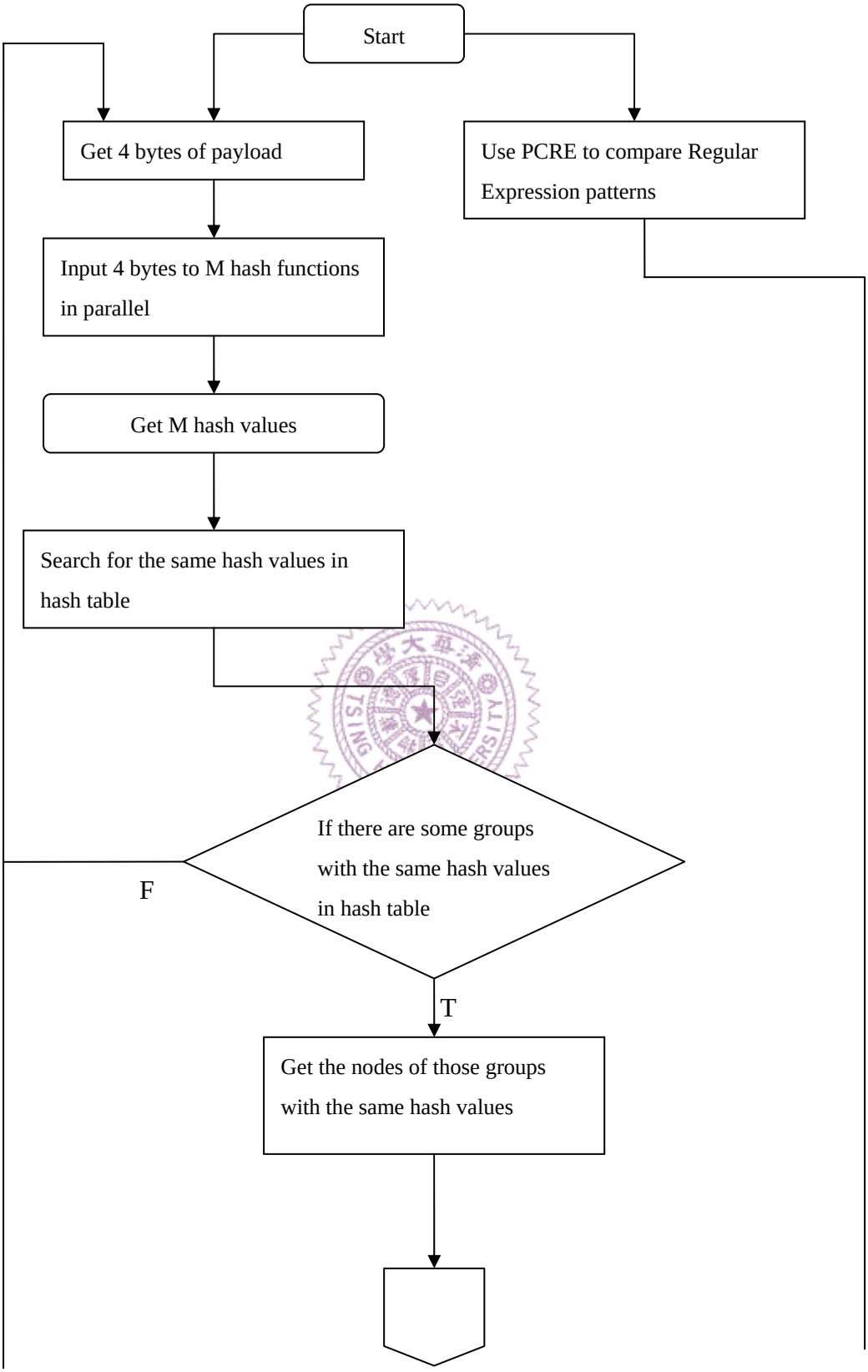
- **Regular Expression Engine**

Because it is not useful to use hash filter and parallel matching engine when processing a Regular Expression pattern, an engine is designed for handling Regular Expression. Actually, PCRE is the most easy and direct way to solve this issue in both Snort and our custom automata system. PCRE handles Regular Expression patterns with single-pattern matching. However, with the more and more complex Regular Expression rules, a Regular Expression automata is also designed in this thesis which will be discussed in Chapter 4.

The flowchart of custom automata system in hardware matching is shown in Figure 3-14.

- **Matching steps in hardware design:**

In the hardware design of our custom automata system, the hash filter is used first to the pass the patterns whose hash values are matched into the matching engine for raising the efficiency of the matching engine. Then with *N P*-bit ALUs in the matching engine, do multi-pattern matching in parallel to achieve high performance. The hardware implementation of custom automata system combines the performance and functionality with low memory requirements.


Start
Get 4 bytes of payload
Use PCRE to compare Regular Expression patterns
Input 4 bytes to M hash functions in parallel
Get M hash values
Search for the same hash values in hash table
If there are some groups with the same hash values in hash table
F
T
Get the nodes of those groups with the same hash values

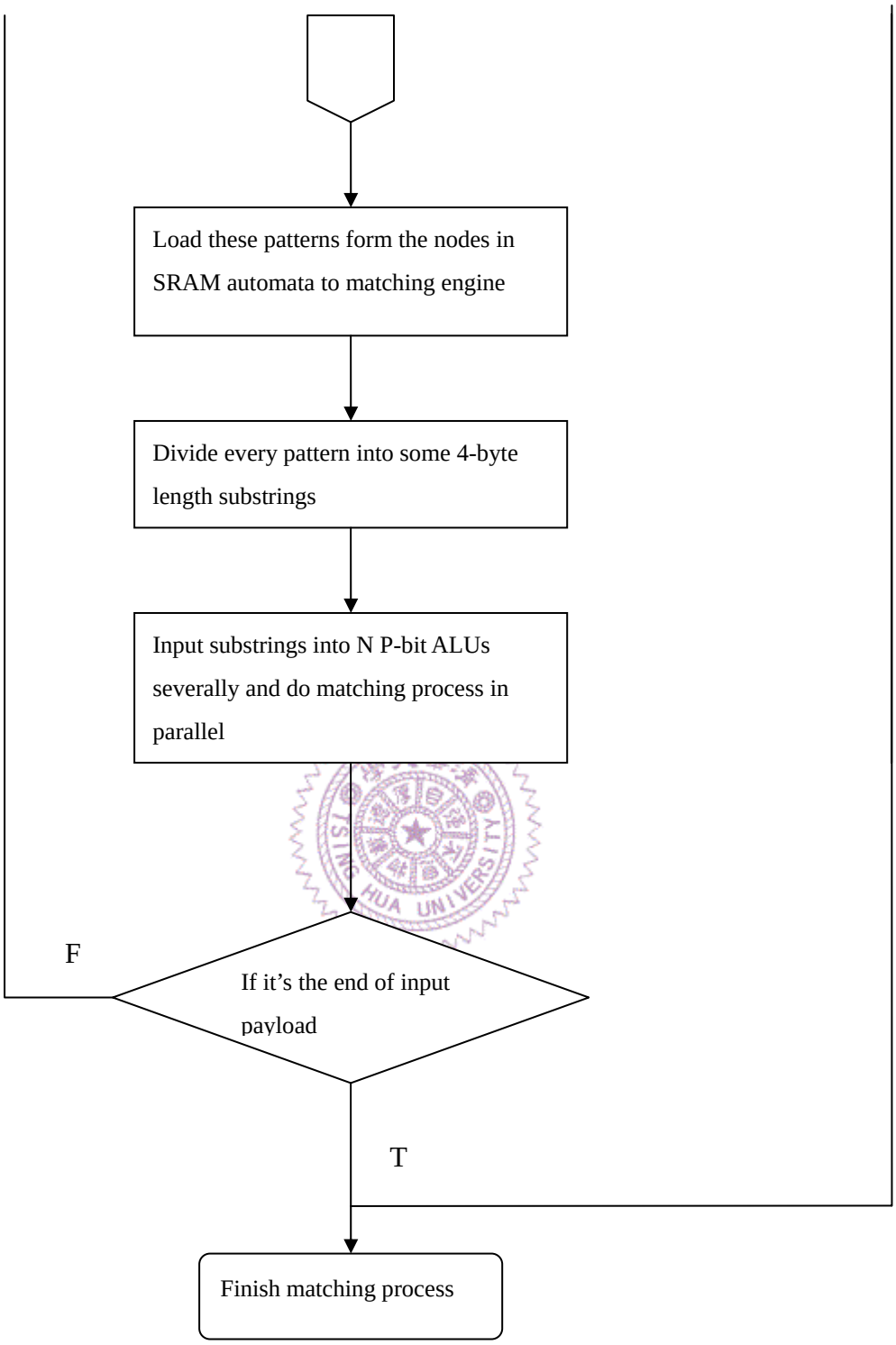


Figure 3-14 Flowchart of custom automata in hardware matching